IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Patent Application of	)
	)
John RHOADES et al.	) Group Art Unit: 2183
	)
Application No.: 10/073,948	) Examiner: Unassigned
	)
Filed: February 14, 2002	)
	)
For: DATA PROCESSING	)
ARCHITECTURES	)

RECEIVED

JUN 1 0 2002

Technology Center 2100

CLAIM FOR CONVENTION PRIORITY

Assistant Commissioner for Patents
Washington, D.C. 20231

Sir:

The benefit of the filing date of the following prior foreign applications in the following foreign counties is hereby requested, and the right of priority provided in 35 U.S.C. § 119 is hereby claimed:

United Kingdom Patent Application No. 0103678.9
Filed: February 14, 2001

United Kingdom Patent Application No. 0103687.0
Filed: February 14, 2001

United Kingdom Patent Application No. 0121790.0
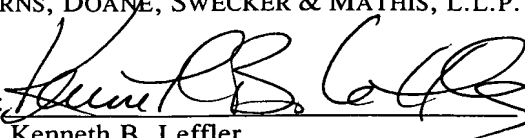Filed: September 10, 2001

In support of this claim, enclosed are certified copies of said prior foreign applications. Said prior foreign applications were referred to in the oath or declaration. Acknowledgment of receipt of the certified copies is requested.

Respectfully submitted,

BURNS, DOANE, SWECKER & MATHIS, L.L.P.

Date: June 6, 2002   By: _____
Kenneth B. Leffler
Registration No. 36,075

P.O. Box 1404
Alexandria, Virginia 22313-1404
(703) 836-6620

(03/01)

THIS PAGE BLANK (USPTO)

CERTIFIED COPY OF PRIORITY DOCUMENT

RECEIVED

JUN 1 0 2002

Technology Center 2100

I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.

Signed

Dated   18 February 2002
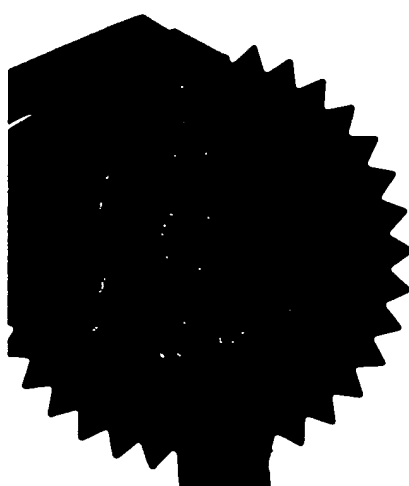
THE PATENT OFFICE
A
14 FEB 2001

The Patent Office

1/77

Request for grant of a patent

(See the notes on the back of this form. You can also get an explanatory leaflet from the Patent Office to help you fill in this form)

14 FEB 2001

1.	Your reference	HL78136/000/CIV

2.	Patent application number *(The Patent Office will fill in this part)*	**0103678.9**

15FEB01 E606239-2 002859
P01/7700 0.00-0103678.9

3. Full name, address and postcode of the or of each applicant *(underline all surnames)*

PIXELFUSION LIMITED
Wallscourt Farm
Filton Road
Bristol BS34 8RB

Patents ADP number *(if you know it)*
If the applicant is a corporate body, give the country/state of its incorporation

O7S1863800 3

United Kingdom

4. Title of the invention
NETWORK PROCESSING

5. Full name of your agent *(if you have one)*

Haseltine Lake & Co.

"Address for service" in the United Kingdom to which all correspondence should be sent *(including the postcode)*

Imperial House
15–19 Kingsway
London WC2B 6UD

Patents ADP number *(if you know it)*

34001

		Country	Priority application number *(if you know it)*	Date of filing *(day/month/year)*
6.	If you are declaring priority from one or more earlier patent applications, give the country and the date of filing of the or of each of these earlier applications and *(if you know it)* the or each application number			

		Number of earlier application	:	Date of filing *(day/month/year)*
7.	If this application is divided or otherwise derived from an earlier UK application, give the number and the filing date of the earlier application			

8.	Is a statement of inventorship and of right to a grant of patent required in support of this request? *(Answer "Yes" if:* *a) any applicant named in part 3 is not an inventor, or* *b) there is an inventor who is not named as an applicant, or* *c) any named applicant is a corporate body.* *See note (d))*	Yes

9. Enter the number of sheets for any of the
    following items you are filing with this form.
    Do not count copies of the same document

    Continuation sheets of this form   -

    Description   76

    Claim(s)   -

    Abstract   -

    Drawing(s)   28 +28

10. If you are also filing any of the following,

    Priority documents   n/a

    Translations of priority documents   n/a

    Statement of inventorship and right
    to a grant of patent *(Patents Form 7/77)*   -

    Request for preliminary examination
    and search *Patents Form 9/77)*   -

    Request for substantive examination
    (Patents Form 10/77)   -

    Any other documents
    (please specify)   -

11.   I/We request the grant of a patent on the basis of this application

    Signature *Haseltine lake &*

    Date
    14 February 2001

12. Name and daytime telephone number of
    person to contact in the United Kingdom   Mr. Chris Vigars   [0117] 9103200

Warning
After an application for a patent has been filed, the Comptroller of the Patent Office will consider whether publication or
communication of the invention should be prohibited or restricted under Section 22 of the Patents Act 1977. You will be informed if it
is necessary to prohibit or restrict your invention in this way. Furthermore, if you live in the United Kingdom, Section 23 of the
Patents Act 1977 stops you from applying for a patent abroad without first getting written permission from the Patent Office unless an
application has been filed at least 6 weeks beforehand in the United Kingdom for a patent for the same invention and either no
direction prohibiting publication or communication has been given, or any such direction has been revoked.

Notes
a)   If you need help to fill in this form or you have any questions, please contact the Patent Office on 0645 500505.
b)   Write your answers in capital letters using black ink or you may type them.
c)   If there is not enough space for all the relevant details on any part of this form, please continue on a separate sheet of paper and
     write "see continuation sheet" in the relevant part(s). Any continuation sheet should be attached to this form.
d)   If you have answered "Yes" Patents Form 7/77 will need to be filed.
e)   Once you have filled in the form you must remember to sign and date it.
f)   For details of the fee and ways to pay please contact the Patent Office.

NETWORK PROCESSING

Section 0

Introduction

The present invention relates to network processing, particularly, but not exclusively, for systems for processing data.packet flows.

In recent years, the proliferation of computers and computer-based equipment has meant that drastic measures have had to be taken to manage and control digital data, in all its various forms, within individual items of equipment and during its transmission between such items. This has become particularly acute with the explosion of communication via radio (for example cellular telephones, satellites, navigation and so on) and of course the Internet.

Packet switching offers a solution to data handling in that individual bundles of data, each associated with its own "header", identifying the data and its destination, and "payload", ie its data content, can be switched, routed and processed in discrete amounts and at ever increasing speeds. Packets of variable length but as large as 65,000 bytes or more can be formed and handled nowadays.

The present invention provides specific solutions to network processing/packet handling in a plethora of devices and systems in this field. Each such solution will be presented in a separate, numbered Section throughout this specification. However, as an aid to presenting the invention clearly and to place the contribution of each such solution in context, the next following section provides a brief overview of the state of the art in this particular field. In addition, it will be appreciated that this art tends to rely on

specific acronyms/jargon to enable concise communication. This specification is no exception. However, the first use of a particular item of directed language, unless already established and understood to be "standard", will be spelled out but thereafter, in the interests of brevity, will be referred to by its acronym. This applies to both text and drawings.

Background

(a) General

The design requirements for individual systems are usually driven by cost or speed or both. Certain protocols have been introduced to enable compatible interface between designers of different items of equipment and/or software. Currently, the commonly adopted industry standard is TCP/IP, which has been mapped onto the OSI model. The OSI model contains seven layers, ranging from the application layer at the top (layer 7) to the physical layer (layer 1) where hardware connection occurs. Each layer has its own protocols. Typical protocols in layer 7, for example, include HTTP for Web, SMTP for e-mail, and FTP for file transfer. The Internet Protocol (IP) in the Network layer, layer 3, uses variable length packets.

Asynchronous Transfer Mode (ATM) is a common protocol in Wide Area Networks (WAN), as opposed to Local Area Networks (LAN), which uses fixed length "cells", as opposed to variable length packets. IP packets can be transmitted via ATM by encapsulation techniques such as IP-over-ATM. The overhead involved is significant. A more efficient solution involves carrying IP traffic over Sonet (Synchronous Optical Networks) technology, known as POS (Packet-over-Sonet). At the hardware level, layers 1 and 2, Ethernet is the most widely used protocol at present, but is liable to

change to accommodate increasing speeds, 10Gb/s. Ethernet interfaces are subdivided into the Media Access Controller (MAC) and the physical layer (PHY). MAC is pure digital logic, operating in layer 2, whereas PHY operates at layer 1.

The network itself consists of three main areas of concern, namely the access area by which communication is effected (cable, wireless etc), the area on one side which defines the environment (web switches, WAN etc) and the service provider area on the other side (router, switch etc). Web switches are a recent innovation designed to support large, complex, distributed web servers, typically in data centres. The service provider area is where Sonet is implemented. The switching function requires intelligence and speed and should aim to provide switching at the fastest speed that the network can handle ("wire speed", currently around 95Mb/s for Fast Ethernet). The access area used to be identifiable by reference to the nature of the medium and was traditionally considered as telecommunications equipment. However, the melding of voice, data, video etc has largely blurred its identity.

(b) Switches and Routers

Router functions were originally performed by a central CPU handling every packet from every port. They consisted of line cards ("blades") plugged into a backplane. Network processors can be applied to line cards in bus-based architectures and thereby take over some of the processing handled by the CPU.

Routers perform two functions, namely control and forwarding. The former can be slower, since payload does not pass through it. The forwarding instruction from the controller is processed in a control plane that operates a look-up table (LUT) in software to direct or

forward the packet ideally at wire speed. In its simplest implementation, only the destination address needs to be examined.

(c) Network processors (NP)

5   The most convenient understanding of what is fast becoming an amalgam of different functionalities and/or architectures is to consider an NP as a programmable device that operates optimally for packet processing. This understanding places NPs apart from general purpose

10   CPUs, since they are not optimised for packets, from chips (such as Ethernet chips), that are not properly programmable, and from co-processors (classification or search engines or traffic managers) since they only handle part of the packet-processing task.

15   The distinct advantage of these processors is that their functionality is comparatively simple. They merely need to extract the required information from the header, process it simply and use it for forwarding. Individual packets in a data stream can be handled

20   efficiently using an array of processors. Stripped down to their bare essentials, these basic packet processors can be dotted around a silicon chip and take up little room (literally only a few square millimetres). One NP chip may contain up to 16 of these stripped down

25   "engines". These engines may moreover be multi-threaded, enabling a process to be conducted on one packet whilst information is being obtained for another.

A typical NP might have four interfaces, enabling connection to the PHY (the line interface), to external

30   switching (the "fabric" interface) or perhaps another NP, to memory, such as destination look-up table memory (LUT) and temporary buffer memory (the memory interface), and to an external host processor (the host interface).

Network processors have distinct advantages over CPUs and ASICs, their main contenders, in that being programmable they are future-proof, at least to a certain extent. For example, changes in standards/protocols may be reprogrammed into the NP. Also, the risk involved in designing an ASIC can be side-stepped by implementing the desired functionality by programming an off-the-shelf NP for which another developer has borne the manufacturing risk.

On the other hand, the programmability of NPs demands increased software development and the newness of these solutions inevitably means that there will be a degree of redundancy in designs/chips that will add to initial costs.

Summary of the Invention

One embodiment of an aspect of the present invention provides a network processor. Its basic architecture is designed to operate in the region of 40Gb/s but is scaleable down to 10Gb/s and up to more than 100Gb/s to meet future needs. The NP is fully compliant with the protocols indicated above. In particular, IP protocol at layer 3, Point-to-Point (PPP), MPLS and ATM for layer 4 and TCP at up to layer 7. However, the NP can support multiple protocols simultaneously. In addition, the NP supports Quality of Service (QoS), enabling multimedia traffic, such as telephony. Packet sizes ranging between 40 bytes and 64 Kbytes can be handled.

Invention Statements

Section 1

In a first aspect, the invention comprises a network processor comprising a series of pipelined SIMD processor units from 1 to n, the units from 1 to n-1 adapted to perform routing functions, and the nth unit

adapted to perform a traffic management function. Preferably, the units 1 to n-1 in the series are of one type whereas the nth unit is of a second type. Each unit of first type preferably comprises an input block,

5   an output block, a processing array between said input and output blocks, and said processing array adapted to communicate with a lookup table unit via an intelligent bus.

According to one aspect of the present invention there

10   is provided a network processor which includes a single instruction multiple data (SIMD) processing core for processing data from a data packet stream.

According to another aspect of the present invention there is provided a network processing device

15   comprising a plurality of network processors each of which includes an SIMD core, the network processors being arranged in series.

Section 2

In a second aspect, the invention comprises a network

20   input processor for a network processor, wherein the network input processor has means for receiving packet data, and means for separating the packet data into a number of smaller chunks of data for processing within the network processor.

25

Section 3

In a third aspect, the invention comprises a network output processor for a network processor, the network output processor having means for receiving a plurality

30   of data chunks, and means for reconstituting the data chunks into a larger packet for onward transmission.

Section 4

In a fourth aspect, the invention comprises a bus structure for a system on (SoC) device comprising a plurality of functional units which transfer data via the bus structure, the bus structure comprising a plurality of unit bus blocks, each of which is connected with a functional unit. Each such unit bus block comprises means for retrieving data packets from an incoming data packet flow for supply to the functional block connected to the unit bus block. Each unit bus block also includes a unit for transferring a data packet to the bus structure from the functional unit connected with that unit bus block.

Section 5
In a fifth aspect, the invention comprises a processing core including all or some of the features described in Section 5 below.

Section 6
In a sixth aspect, the invention comprises a lookup table which comprises a plurality of value storage levels, the storage levels being accessible via input key values. According to another embodiment of this aspect of the present invention, there is provided a method of performing LC trie lookups to produce table lookup results.

Section 7
In a seventh aspect, the invention comprises a semaphore unit comprising all or some of the features as described in Section 7 below.

Section 8

In an eighth aspect, the invention comprises a method of processing a data packet stream including all or some of the features set out in section 8 below.

SECTION 1

Architecture

5   Figure 1.1 illustrates a network router which
includes a network processor which embodies at least
one aspect of the present invention. The network
processor (NP) receives data from a physical layer
(PHY) and outputs data to a switch fabric of the
10   router. The switch fabric operates to route data and
data packets to a correct output port of the router in
dependence upon switching information provided by the
network processor.
         The NP includes a network input processor (NIP)
15   which is connected (in the ingress path) to receive
data in the form of a stream of data packets from the
physical layer PHY. The NP also includes an embedded
processor unit EPU, and control plane interface unit
CPI for interfacing with a control plane of the router.
20   Data processing on the NP is performed using a single
instruction multiple data (SIMD) processor core
structure. A SIMD processor core includes a plurality
of processing elements that operate in parallel. All
of the processing elements on a processing block carry
25   out the same single instruction at a given point in
time, but operate on respective data portions. The
structure of the processing core will be described in
more detail below.
         The SIMD processor core receives data from the NIP
30   and can communicate data with the remaining units in
the NP. A lookup table unit LUT is used for storing
various lookup data. A network output processor NOP is
used to transfer data from the NP to the switch fabric
of the router. The units in the NP transfer data to
35   the other units on the NP using a bus structure. In
addition, external memory, such as static random access

memory (SRAM), can be connected to the bus structure to provide increased storage facility for the NP.

Instead of using a single network processor NP to provide all of the processing power required for one
5   input port of the router, it is possible to pipeline a number of NPs together. Figure 1.2 illustrates such a pipeline of network processors. The network processors NPA, NPB, NPC and NPD are arranged such that processor NPA receives data packets from the physical layer PHY.
10   Each of the following NPs receives data packets via its NIP from the NOP of the previous NP in the pipeline. Each network processor performs its predefined processing steps on the received data packets, and then outputs the processed data packets to the next network
15   processor in the pipeline. Each NP can be optimised to perform a particular function, which would enable the overall function of the processing to be carried out in a more efficient manner than using a single NP.

According to one aspect of the present invention
20   there is provided a network processor which includes a single instruction multiple data (SIMD) processing core for processing data from a data packet stream.

According to another aspect of the present invention there is provided a network processing device
25   comprising a plurality of network processors each of which includes an SIMD core, the network processors being arranged in series.

The SIMD processing core comprises a plurality of processing elements which are arranged to receive
30   respective data portions and to process those data portions in accordance with a single instruction stream provided by a controller. All of the processing elements carry out the same instruction, but on different respective data portions. In one example
35   SIMD structure, the plurality of processing elements are split into a plurality of processing blocks or

mini-cores. Each core structure can be made up of a number of mini-cores. Such a mini-core structure has the advantage that data input/output (i/o) tasks can be performed for each mini-core in turn, and in the

5   meantime the other mini-cores can be controlled to perform useful processing on the data already held in the processing element memories. The mini-cores preferably operate on independent instruction streams, or on one instruction stream that is time shifted for

10 each mini-core. This arrangement can give increased flexibility and speed of processing by allowing operations to be run effectively in parallel between different mini-cores.

The NP architecture also preferably includes an
15 embedded processor unit (EPU), and a control plane interface (CPI).

Figure 1.3 illustrates that a pipeline of network processors can be used in the egress path for outputting data packets to the physical layer from the
20 switch fabric of the router.

These and further features of various embodiments of various aspects of the present invention can be found in the attached annexes.

Section 2 - NETWORK INPUT PROCESSOR (NIP)

The network input processor (NIP) is the interface to the outside world, for example an external physical layer or a network output processor (NOP) of another network processor.

Referring to Figure 2.1, the NIP comprises a Generic I/O interface (shown as the GIO input interface) which receives packet information from a Generic I/O block, GIO, (not shown). The packets can be of differing protocols, such as IPV4, IPV6, MPLS, ATM, and so on. Although the preferred embodiment shows the GIO as a separate entity from the NIP, it is noted that the GIO could also form part of the NIP itself, or part of the network processor.

The packet information received by the GIO input interface typically comprises the raw data (or payload), information relating to the data length (for example as extracted from the layer 2 framing information), data type and other control information. Figure 2.2 shows a typical packet in which the packet header contains the information concerning the data length, type and other control information, with the raw data provided in the packet payload (PL).

The GIO input interface adds a prepend control header to the incoming packet, before passing the packet to a packet buffer which, in simple terms, is a FIFO.

A NIP state machine receives packet data from the packet buffer, and performs tasks on the packet data, which may be under hardware or software control. Although the NIP state machine is shown in the data

path, it should be noted that the NIP state machine provides the function of extracting data from the data path, operating on the data, and feeding the data back into the data path.

If the packet in the packet buffer is less than a predetermined length, the NIP state machine will retrieve the packet from the packet buffer, update the prepend control header and forward the packet to the NIP-to-Bus output interface.

If the packet is greater than a predetermined length, the NIP state machine will divide the packet into smaller packets, known as chunks, and add a prepend control header to each chunk. This enables a large packet to be processed in parallel by the processing elements of a SIMD array. The chunks generated by the NIP state machine are forwarded via the NIP-to-Bus output interface to their appropriate destination, for example the SIMD processing array for processing.

Figure 2.3 shows how the incoming packet may be divided into a number of chunks, each chunk having a prepend control header (CH).

In addition to receiving packet streams from the GIO input interface and packet buffer, the NIP state machine can also receive packet information from within the network processor itself. For example, the NIP can receive data which has been sent back from the SIMD array or from the NOP. Data can also be received from other elements, such as the control plane, EPU or other module, for example a "proxy NOP".

The network processor can determine whether recycled data is returned to the NIP in the form of packets or chunks. The accepting of packets or chunks back from the processing array, NOP, or some other part

5   of the network processor, is referred to hereinafter as recycling. The data could be in chunk format when returned to the NIP, for example if received directly from the SIMD array, or the data could be in packet format, for example if returned from the NOP having

10   been de-chunked.

The NIP comprises a NIP-to-Bus input interface which receives recycled chunks or packets from the bus, and passes the chunks or packets to a de-chunked

15   buffer. The packets or chunks sit in the de-chunked buffer until the NIP state machine decides to read them from the de-chunked buffer.

If chunks have been passed back to the NIP, for

20   example from the array of processing elements, the NIP state machine may decide not to perform any chunking, and may merely update the prepend control header of each chunk before routing it via the NIP-to-Bus output interface to the required destination.

25

If packets greater than a predetermined length have been passed back to the NIP, then the NIP state machine may decide to perform chunking again, add a prepend control header to each chunk, and route the

30   chunk via the NIP-to-Bus output interface to the required destination.

Recycling has the advantage of enabling chunks or sets of chunks to be processed more than once. For

35   example, if the processing array is unable to process a chunk or packet in a given time, it can mark the data

as a recycle chunk or packet for recycling back to the
NIP. This avoids the processing array having to wait
for a particular task to be completed, which would
otherwise delay the other data being processed.

5

Figure 2.4 shows an example of a prepend control
header (CH) according to one aspect of the invention.
It is the prepend control header that governs the
processing of a packet or chunk within the network
10   processor or processors.

The prepend control header comprises a "load
address" field which is configurable under software
control. The load address informs the load engine of
15   the SIMD array where the chunk should be loaded in the
processing elements' memory. The load address may
change depending on whether the chunk is being
processed for the first time from the GIO input
interface, or whether the chunk is being recycled. In
20   the latter case, the address is taken from the control
register.

The prepend control header also contains
information passed by the layer 2 protocol of the
25   incoming packet, in particular concerning layer 3
protocol data. This information will typically
indicate whether the data is an IPV4, IPV6, MPLS, ATM
or similar type of data, and the length of the data
packet.

30

The "chunk identifier" field is used to identify
the sequence number of the chunk, and the "length of
chunk" field is used to indicate the size of the chunk,
excluding the prepended control header.

35

The prepend control header also comprises a number of flags. For example:

- the "C" flag is used to indicate whether that particular chunk has been chunked or not

- the "F" flag is used to identify the final chunk in a set of chunks

- the "R" flag is used to indicate that the chunk is a recycled chunk

- the "E" flag is used to indicate that the chunk has an error, or is part of a larger packet in which one of the chunks has an error

- the "P" flag is used to indicate that the chunk should be passed to the control plane

- the "D" flag is used to route the chunk directly to the NOP for sending downstream, i.e. by-passing the processing array.

- Other flags, for example length of payload, eg. software reserved bits.

When a packet is divided by the NIP state machine into a number of chunks, the first chunk of the packet will have the "C" flag set to indicate that it has been chunked. The "F" flag will not be set because it is not the final chunk. Subsequent chunks will have their "C" and "F" flags set in the same way. The final chunk of the packet will have both the "C" flag and the "F" flag set, thereby indicating that it has been chunked and that it relates to the last chunk of that particular packet.

It is noted that the last chunk in a packet may have a different chunk length to the preceding chunks from the same packet. This is because the preferred embodiment does not use any padding for making the final chunk the same length as the other chunks. i.e. it relies on the chunk length to indicate the size of the chunk.

If the "E" flag is set indicating an error, for example by the software running on the SIMD array, the NIP state machine can choose to drop the packet or send it to the control plane for further processing.

The NIP can also be programmed to perform course-grain classification. For example, based on the layer 2 information in the incoming packet, the NIP can decide whether to mark a chunk, and subsequent chunks from a particular packet, as errant or special in some way. The NIP can therefore be programmed to perform additional tasks, such as dropping chunks relating to a particular packet having errors, or passing chunks to a control plane for further processing.

The NIP can be programmed to forward data in chunk form or in packet form as they were originally received. According to the latter case, the NIP state machine merely routes packets to the control plane without dividing the data into smaller chunks.

The NIP provides a distributor function whereby the chunks are passed from the NIP to the processing array in sequence. This may be controlled by the NIP if it knows the sequence ordering. Alternatively, the processing elements can request data from the NIP in the order in which they should be loaded. Both these options are very powerful since they avoid the need to re-order or re-sequence the data at a later stage.

Thus, it is preferable that, when the NIP state machine changes between receiving data from the packet buffer and de-chunked buffer, it does so at a packet boundary, thereby maintaining packet order.

The NIP comprises a number of registers, some of which can be used to change the behaviour or configure the behaviour of the NIP, and some of which can be used as counters for maintaining information relating to the data being processed by the NIP.

For example, the counters can be used to indicate the number of packets that have been marked as errant, the number of packets discarded, the number of packets transferred to/from the control plane, the number of packets passed downstream, and so on.

It is noted that the predetermined chunk size or length mentioned above can be varied under software control, for example in accordance with a particular application or the type of data being received.

The NIP may also comprise additional buffers, for example FIFO buffers, between the NIP state machine and the NIP-to-Bus output interface, and/or the NIP-to-Bus input interface and the De-chunked buffer. These buffers function to de-couple what is travelling on the bus and the behaviour at the interfaces with the bus.

According to the invention described in this section, there is provided a network input processor for a network processor, wherein the network input processor has means for receiving packet data, and means for separating the packet data into a number of smaller chunks of data for processing within the network processor.

Preferably, the network input processor adds information to each chunk to control its flow and/or processing through the system.

Preferably, the packet data is separated into a number of chunks for processing in parallel in a plurality of processing elements.

5   Preferably the network input processor further comprises means for receiving data which has been recycled from within the network processor.

10   According to another aspect of the invention, there is provided a prepend control header for controlling the processing of a data chunk within a network processor.

15   Preferably the prepend control header comprises a flag for indicating whether a particular chunk forms part of a larger data packet.

Preferably the prepend control header comprises a flag for identifying the final chunk in a series of
20   chunks.

Preferably the prepend control header comprises a flag for indicating that a chunk is a recycled chunk.

25   Preferably the prepend control header comprises a flag for indicating that a chunk contains an error, or is part of a larger packet having an error.

Preferably the prepend control header comprises a
30   flag for indicating that the chunk should be passed to a control plane in the network processor.

Preferably the prepend control header comprises a flag for indicating that a chunk should bypass the
35   processing elements in the network processor.

These and further features of various embodiments of various aspects of the present invention can be found in the attached annexes.

Section 3 - NETWORK OUTPUT PROCESSOR (NOP)

The network output processor (NOP) is the partner
to the NIP in the sense that it reconstitutes the
5   original incoming packet for onward transmission, for
example to the outside world, or back to the NIP or
another module in the network processor for further
processing.

10   It is noted that there are occasions where the NOP
does not perform the function of reconstituting the
incoming packet, for example, when a number of similar
network processors are pipelined, in which case it may
be desirable to preserve chunks through each stage of
15   the pipeline. This may be provided as a software
configured option.

The NOP can receive its data from a number of
sources within the network processor. For example,
20   data may be received from the SIMD array of processing
elements, directly from the NIP, or from other elements
of the network processor.

In one mode of operation, the NOP receives chunks
25   from the SIMD array of processing elements. The chunks
may or may not be the original chunks that the NIP sent
out to the processing elements, for example the control
plane can inject new packets.

30   The NOP will preferably receive chunks in the same
order as they were delivered from the NIP to the
processing elements. This has the advantage of
preserving packet order, thereby enabling packets to be
reconstituted with minimal computing overhead.

35

Referring to Figure 3.1, the NOP comprises a NOP-to-Bus input interface for receiving data. Data is passed from the NOP-to-Bus input interface to the NOP state machine. Although the NOP state machine is shown

5   in the data path, its function is to extract data from the data path, operate on the data, and feed the data back into the data path. The data from the NOP state machine is fed to either the packet buffer or the recycling and exception buffer. The function it

10   performs will depend on the information contained in the prepend control header.

Figure 3.2 shows a typical set of chunks which may be received by the NOP state machine. Each chunk

15   contains a prepend control header, or chunk header (CH). In addition, the first chunk in the set comprises a prepend header "PreH", which is the result of the previous computing tasks. For example, the PreH header could provide information relating to:

20

    –   the output queue ID to which the packet should be sent

    –   results of searching within the packet payload, or

25   –   control information to allow the packet to be reprocessed.

There are basically two data paths through the NOP. The first comprises sending reconstituted chunks

30   or packets from the NOP state machine via a packet buffer to a GIO output interface for onward transmission. The reconstitution takes place on-the-fly, i.e. between the NOP state machine and the packet buffer.

35

Figure 3.3 shows a typical packet having been reconstituted. The prepend control header (CH) has been stripped from each chunk, and used to create the outgoing packet. The outgoing packet comprises the

5   original payload and header, but also comprises the additional prepend header, PreH, which is used to control further processing or routing of the packet.

The second possible path through the NOP involves

10   routing the reconstituted chunks or packets from the NOP state machine to a Recycling and Exception buffer. Data in the Recycling and Exception buffer is passed to a NOP-to-Bus output interface for transmission to other modules within the network processor.

15

Thus, depending upon the status of the various flags in the prepend control header, the NOP state machine can choose to route data via either of these paths.

20

The NOP state machine can also decide whether or not to de-chunk data it receives from the NOP-to-Bus input interface. For example, if the prepend control header indicates that a set of chunks belong to the

25   same packet, the NOP state machine can reconstitute these chunks into the original packet for outputting via the packet buffer to the GIO output interface, as described above.

30   Alternatively, chunks or packets passed to the NOP-to-Bus output interface can be recycled within the network processor to the NIP.

Alternatively, the chunks or packets can be sent

35   via the NOP-to-Bus output interface to the CPI. The CPI controls the flow of data to and from the control

plane. This can be advantageous if a set of chunks or packets require further processing, for example for billing purposes.

5   The NOP state machine will determine whether or not to de-chunk the received data depending upon whether or not the "C" flag has been set in the prepend control header (CH). The "F" flag is used to identify the final chunk for reconstituting the original packet.

10

The "R" flag is used to indicate that a chunk should be recycled, in which case the chunk or packet is sent to the Recycling and Exception buffer.

15   Similarly, if the E" flag is set, the NOP state machine will know the chunk contains an error, and can choose to discard the chunk, or, alternatively, it can pass the errant chunk to the Recycling and Exception buffer, for onward transmission to the control plane
20   for further investigation.

The P" flag is used to indicate that a chunk should be passed to the control plane, for example if the chunk contains an error as mentioned above, or if
25   the chunk is more suited for processing at the control plane.

It is preferable that, when the NOP state machine changes between passing data to the packet buffer and
30   Recycling and Exception buffer, it does so at a packet boundary, thereby maintaining packet order.

The function of the recycling part of the Recycling and Exception buffer is to deal with chunks
35   or packets which are to be recycled. The function of the exception part of the Recycling and Exception

buffer is to deal with exception. For example, if it is determined that there is something wrong with a given packet, a normal packet could be reconstituted for onward transmission, plus an additional exception

5   packet which would be sent to the Recycling and Exception buffer for further processing elsewhere.

The NOP comprises a number of counters for maintaining information, similar to those provided in

10   the NIP. For example, the counters can be used to indicate the number of packets that have been marked as errant, the number of packets discarded, the number of packets transferred to/from the control plane, the number of packets passed downstream, and so on.

15

The NOP also provides a collector function. This is a mode of operation whereby the NOP collects chunks from the processing elements of the SIMD aray. The order in which chunks are collected can be sequenced by

20   software running on the processor array.

The NOP may also comprise additional buffers, for example FIFO buffers, between the NOP-to-Bus input interface and NOP state machine, and/or the Recycling

25   and Exception buffer and the NOP-to-Bus output interface. These buffers function to de-couple what is travelling on the bus and the behaviour at the interfaces with the bus.

30   To summarise, the functions that the NOP may perform are as follows:

              -    Accepts chunks from compute engines in the
                   SIMD array(s) sequentially. The chunks are

35   received in a deterministic manner, for
                   example timing, order and number.

- De-chunks by making use of the prepend header. If this option in enabled in the NOP, the original packet will be recreated by recombining the chunks. This is done by extracting information from the prepend control header to recreate the original packet stream.
- Optionally forwards exception packets to the control plane via the Control Plane Interface (CPI). Any chunks that the software executing on the SIMD array(s) has indicated as exceptional are forwarded to the control plane.
- Optionally forwards packets to the NIP. Any chunks that the software executing on the SIMD array(s) has indicated as such are forwarded to the NIP. It is noted that, for certain chunks, this can happen in addition to forwarding to the CPI. For example, if the "P" and "R" flags of the prepend control header are set, a chunk or packet is sent to the control plane and recycled.
- Maintains counters, for example, the number of packets, number of chunks, number of dropped packets, etc.
- Accepts packets from the control plane via the CPI. This mode allows for the control plane processor to inject a sequence of packets or chunks into the switch fabric or to the next stage in a pipelined network processor. The NOP has means for dealing with the flow of data from the control plane and SIMD array. For example, additional FIFOs (FIFOX in Figure 3.1) could be included to buffer data from the control plane and SIMD array, respectively. The additional

FIFOs, FIFOX, enable the NOP state machine to change from processing SIMD array data to control plane data at packet boundaries, thereby maintaining chunk order. It is noted that this function may be realised in other ways.

- The collector function provides a mode of operation where the software executing upon the array of compute engines can direct the chunk order it wishes to send.

According to the invention described in this section, there is provided a network output processor for a network processor, the network output processor having means for receiving a plurality of data chunks, and means for reconstituting the data chunks into a larger packet for onward transmission.

Preferably, the network output processor further comprises means for recycling data chunks to other parts of the network processor.

These and further features of various embodiments of various aspects of the present invention can be found in the attached annexes.

SECTION 4.0 - BUS

Figure 4.1 illustrates a network processor
embodying at least one aspect of the present invention.
The processor includes a network input processor $\overline{NIP}$, a
SIMD processing core, a look up table LUT and a network
output processor NOP. The network processor may
include additional functional units, as required. The
functional units are interconnected by way of a bus
structure.

Figure 4.2 is a simplified schematic diagram
showing the bus structure in more detail. The bus
structure is made up of a number of bus modules, or FB2
blocks. The number of FB2 blocks provided is dependent
upon the length of bus required and on the number of
functional blocks to be connected to the bus. Each FB2
block has a single functional block connected
therewith. In the simplest form of bus structure, each
functional block receives data from and supplies data
to the bus via a single FB2 block.

The FB2 blocks are interconnected to one another
in a series to form the bus structure. The
interconnections, or "bus lanes", are preferably
unidirectional and multiple. In order to achieve
desirably high bandwidth, the bus structure could
include a single very wide bus. However, this would
not be practical. Instead of implementing a single,
very wide bus, several unidirectional buses will be
implemented, operating in parallel. This provides
additional quality of service control, as access to
each of the bus lanes can be prioritized. Preferably,
there are an equal number of bus lanes operating in
each direction, although the exact configuration of bus
lanes is flexible and depends on the application in
which the bus is to be used. Each FB2 block includes a
bus lane module for each bus lane supported by the FB2

block. In order to supply the required bandwidth, multiple buses are required. The proposed approach assumes that up to 4 128-bit wide lanes in each direction are wired into each block.

5   Figure 4.3 illustrates one such module contained in the bus module FB2. The bus lane (incoming bus lane) supplies data to the bus lane module which operates to determine whether the incoming data is for supply to the functional block connected

10   with the FB2 block concerned. The bus lane module comprises a store unit which receives incoming data and buffers that data if required. A consume controller operates to transfer incoming data from the bus lane to the functional block connected with the FB2 block, and

15   an inject controller operates to control supply of data from the functional block to the bus. A multiplexer MUX supplies an output bus lane with data from the bus lane module, either data direct from the store unit or from the functional block concerned.

20   In order to explain the operation of the FB2 block, the data flow along the bus will be explained. The bus has a predefined width, which determines how many bit of data can be transported in parallel along the bus. Typically the data intended for a functional

25   unit will have more bits than can be transferred in parallel at the same time along the bus. This is particularly the case for packetised data flows in which data is transferred between units in discrete packets. In the case of data packet flows, it is

30   important that an entire packet of data is transferred to the functional block attached to the FB2 block.

         Figure 4.4 illustrates data flow along the bus. Since the data cannot all travel along the bus in parallel, it is divided into a number of flow control

35   digits, or "flits". Figure 4.4 illustrates single uni-

directional bus lane, but the principles can be extended to cover other arrangements of bus structure.

Figure 4.5 illustrates a number of flits passing along the bus. A head and tail bit are used to mark the start and finish of the data packet.

The bus uses a technique known as wormhole routing. The bus is made up of a number of interconnected logic blocks. Figure 4.5 illustrates wormhole routing. Ignore for the moment the interfaces onto and off of the bus. The bus is made up of a number of connected logic blocks. Each block contains flit registers. The flits that make up each packet act like the wagons of a train, following one another from block to block. The front (header) flit of each packet contains control information that is used to direct the path of the packet along the bus. This control information includes identity information for the functional unit for which the data packet is intended. Using the train analogy, the header flit acts like the engine of the train, pulling the remaining flits along the bus (network).

Packets travelling along the bus pass through a number of intermediate interface blocks. Preferably, each block should introduce no more than on cycle of latency, so that overall latency is not too great. Each flit should only experience one clock cycle of delay if there is no contention on the bus. This can present a problem if only one flit register is provided in each store block. When a packet header gets blocked, the "halt" signal needs to be sent to all of the flits behind it. This cannot be done in one clock cycle. To overcome this, each block must use two registers with muxes and de-muxes (see Figure 4.6). Flits can pass through in one cycle, and when the header is blocked, the trailing flits are able to queue behind it.

Figure 4.7 shows the situation when a functional block injects a new packet onto the bus. The functional block generates a control signal to the inject controller of the FB2 block to which it is

5   connected. The inject controller causes the store block to prevent travel of the leading flit of the next data packet into the FB2 block. An acknowledgement signal is then returned to the functional block to cause the functional block to enable the block to

10   commence transmission of its data packet. The outgoing data packet is supplied through the inject controller and the MUX to the bus lane concerned for transport down the bus to the next FB2 block.

Figure 4.8 shows the situation when a header flit

15   arrives in the FB2 block (which header flit marks the start of a data packet) is intended for the functional block connected with the FB2 block concerned. The leading (header) flit is received by the store block, and its address or module ID field is examined to

20   determine if the flit is intended for the functional block concerned. In the present example it is assumed that the flit is to be transferred to the functional block, and so a control signal is sent to the consume controller. The consume controller indicates to the

25   functional block that a data packet is to be received and when the functional block indicates that it can receive the data packet the packet is transferred, flit by flit, to the functional block.

Figure 4.8 also illustrates that while the

30   functional block is receiving a data packet, it is possible for the block to output a data packet onto the bus. Since the flow along the bus lane concerned has been temporarily stopped by the removal of a packet to a functional unit, that functional unit can inject its

35   own data packet onto the bus lane.

If the store unit receives a new flit that it is unable to forward, either along the bus lane or to the functional block connected with the FB2 block concerned, the flit is held temporarily in the store

5   block's buffers.

In previously-considered bus structures, the injection of data onto the structure has been controlled by a central arbitration unit or arbiter. In the present bus structure, arbitration is provided

10   at each FB2 block. The distribution of the arbitration functionality means that the individual arbitration scheme can be relatively simple. In the present example, the inject controller determines if a data packet is to be injected onto the bus, and will simply

15   instruct the store block of the FB2 block to halt the next leading flit, to enable a new data packet to be placed on the bus. This distributed arbitration scheme means that the bus structure is scalable and can simply be tailored to the specific application requirements by

20   adding as many FB2 block as required.

Such a structure also enables high speed data transfer to occur, because only a simple comparison needs to be made between an incoming leading flit and the ID of the functional block to determine if the flit

25   is intended for the functional block or if it is to be forwarded.

The use of multiple store blocks means that data signals are driven shorter distances than on a traditional bus structure, where data has to be driven

30   the entire length of the bus. The short distances between the FB2 blocks in a bus structure embodying the present invention enable higher speed operation. This higher speed operation leads to higher bandwidth capability.

35   The use of multiple bus lanes allows bandwidth to be allocated to functional blocks according to which

lanes they are allowed to use. If multiple functional blocks communicate on the same lane(s), there needs to be a mechanism to sub-divide bandwidth. The proposed solution to this is to give each block a programmable register and a counter, which controls the number of cycles that the module is allowed to inject its own packets, and the number of cycles for which it must yield to data already on the bus. This requires a global approach to programming the registers, which should be done by an embedded processing unit (EPU) at system initialisation.

This programming of the inject controllers across the bus structure means that bandwidth allocation, and hence quality of service controls, can be actively adjusted by the overall control processor or other controller. Alternatively, in systems which do not include a specific control processor, the registers can be programmed by an authorised controller.

In the bus structure being described, control signals are supplied along the bus lanes as part of the flits being transferred. There are no separate control lines provided for, for example, reconfiguring the inject controllers. This leads to simplified bus structure as well as saving on silicon area used for transfer of such signals.

The overall bandwidth of the bus structure can be greater than the raw bandwidth of the bus, due to concurrent communication. An example is shown in Figure 4.9. While data is being sent from functional block A to functional block B, packets are also being sent from functional block C to functional block D. In this simple example, the throughput is twice the raw bandwidth of the bus. Of course, it is not always possible to exploit such concurrency fully.

The use of concurrency can boost the bandwidth of the bus structure considerably, and will be determined

by the actual bus topology to be implemented in any given application.

The basic bus blocks described so far only allow simple bus topologies, without any branches. At least 5 one other type of bus block is being considered. Perhaps the simplest option is the three way switch. Implementing a switch will allow more complicated bus topologies. One restriction on the topology is that it cannot contain any loops. Loops will introduce 10 deadlock problems, and coping with deadlock would significantly complicate the bus control logic.

One possible enhancement to the bus structure is to provide clock distribution alongside data distribution.

15 In current system-on-chip (SOC) designs, clock signals are routed to the various units of the IC, and optimising the routing can be a complex task. It is necessary to ensure that the delays experienced by the various blocks of the system do not become so large as 20 to cause timing errors between the blocks. In previous systems this has been achieved by adjusting the delays experienced by the blocks in the system so that there is no or very little relative delay between the clock signals arriving at the various functional blocks.

25 A proposed method for distributing the clock is to run the clock along the bus in one direction. Each FB2 block and each SoC will have a local clock. All of these clock trees will be balanced so that they have the same insertion delay rate. The scheme is 30 illustrated in Figure 4.10.

Each SoC block/FB2 block pair will see the same clock, and will have minimal skew relative to one another (for example) < 100 ps). However, the clocks seen at adjacent blocks will be skewed relative to one 35 another by some amount, which is dependent on the distance between the insertion points.

Figure 4.11 shows the relative phases of the clock at points A, B and C along the bus. When the data between bus clocks is traveling in the opposite direction to the clock, the effective clock period is reduced by tskew. Provided tskew is kept small, this should not create any problems. When the data is traveling in the opposite direction, there is a potential for hold-time violations. Provided that there is sufficient logic delay between the blocks, the skew can be masked, and no timing violations will occur. Each bus block is identical, so ensuring that the logic delay in each block is sufficient should not be too difficult.

The datapath delay between adjacent bus blocks must be controlled within upper and lower bounds, to prevent set-up and hold violations. The logic delay has a fixed and well defined upper and lower bound, since each block contains identical logic. The wire delay can be controlled by using bus repeater blocks. These are blocks that simply register the data, but do not contain a Fuzion Bus Interface (FBI) to a block.

Such a scheme can have several benefits, a non-exhaustive lost of which is presented here:

- Clock smearing. May help with power distribution/electromigration.
- The same approach can be applied consistently. Effort is concentrated on characterizing bus block.
- Lower power? Don't need to add extra clock buffers to balance skew globally.
- The clock distribution logic can be built into the bus block macros, offering a simple tiled approach.

It will be clear that there are several advantages of the described bus structure over previously

considered structures. A non-exhaustive list of these
advantages is given below:

- Re-usability

5
- Extensibility-physical placement not
  constrained by bus, bandwidth can be added
  in.

- Programmability

10
- Distributed arbitration

- Control paths are incorporated in bus.

15
- Variable packet length

- Concurrent communication

Using a single bus structure which has no
20 customised signal lines, but rather a standardized data
packet transfer system, means that there is a uniform
wiring structure between FB2 blocks. The data lines
need only be driven for short distances (between the
FB2 blocks) and only when required. This can lead to
25 useful power consumption savings. In addition,
providing a plurality of identical data lines means
that the FB2 block can be made up of a series of
identical logic portions rather than requiring complex
task-specific logic.
30 These and further features of the various
embodiments of various aspects of the invention can be
found in the attached annexes.
According to this aspect of the present invention,
there is provided a bus structure for a system on (SoC)
35 device comprising a plurality of functional units which
transfer data via the bus structure, the bus structure

comprising a plurality of unit bus blocks, each of
which is connected with a functional unit. Each such
unit bus block comprises means for retrieving data
packets from an incoming data packet flow for supply to

5   the functional block connected to the unit bus block.
Each unit bus block also includes a unit for
transferring a data packet to the bus structure from
the functional unit connected with that unit bus block.

SECTION 5 - THE CORE

This is the heart of the processor. The
embodiment of the invention is a radical departure from
5   previous attempts at constructing SIMD processors.
Previously, they have been considered to be one-off
devices. In the present invention, a "modular"
approach has led to the evolution of a SIMD processor
with the capacity for varying the number and
10   functionality of various units, such as processor
elements (PEs), controllers, I/Os, software libraries
and tasks and support that is generic. It allows data
to be ported back and forward between various blocks
and units. This will become clearer as the particular
15   description of the core progresses in this Section.
Figure 5.1 is the overall block schematic diagram
of the SIMD core and its associated controllers and IO
structures. As shown, in its most fundamental
elements, it consists of a Thread Sequence Controller
20   (TSC), a Generic SIMD Controller (GSC), a variety of IO
blocks, the PE blocks themselves, and the Fuzion bus.
It is convenient to consider each of these blocks in
turn.
Figure 5.2 is a block diagram of the TSC. The
25   fetch unit is responsible for taking in instructions.
It can be tailored to the style of expected instruction
stream. If undue quantities of linear code would
result, the fetch unit can be made to "look ahead" of
the current position. Where branching is a major
30   consideration, the fetch unit can pre-fetch the
relevant branches that will be encountered. Likewise,
it can fetch for multiple threads and keep the next set
of instructions for each thread in local cache or other
dedicated memory. More detail of the fetch unit is
35   given later in this Section.

The semaphore unit is the handshaking unit with the rest of the system. Semaphores are broken into groups according to which execution units the semaphore can be signalled by (eg processor elements or

5 controllers). They are the focal point for synchronization of threads and control of resources.

The scheduler uses a set of rules, based on priority and state, in order to determine which thread should be running. In effect, it performs the parallel

10 part of the TSC in this respect. The thread processor in itself is effectively responsible for unwinding the code, which it does in linear fashion. The thread processor applies loops etc to the whole array of PEs as part of the thread.

15 The scheduler decides on the priority to be allocated to each thread and issues the appropriate instruction in parallel fashion across the threads.

In effect, the thread processor comprises a full blown processor. It has the ability to load and store

20 data to and from other devices. It also avoids the extreme complication involved in application to a compiler.

The concept (in Fig. 5.1) of using a mono-processor in the TSC and a poly-processor in the SIMD

25 part, which relates to the processing elements at the bottom of the Figure, enables the processor to operate in this way.

Referring now to the Generic SIMD Controller (GSC) in Figure 5.1, the thread sequence controller will have

30 one controlling aspect per segment of the controller and the objective is to make all those controlling segments appear the same. Referring to Figure 5.3, a pipeline view of the segment is shown. At the bottom, above the core, is the functionality of the controller

35 that targets the specific function in the core or a specific coprocessor function, or a specific hand

shaking function that has to operate at this level within the core. This functional unit sits above the so-called "issue point". Once the issue point is passed, there is a fixed set of delays to and from the

5 execution point in the PE array.

Above this layer of the function there is a hand-shaking port to other segments. A standard interface is used here that can communicate with any other segment so that, if a segment wants to take control, or

10 "hijack", some other function, it can do so with impunity. The segment that is the BIO controller can request load/store PE access via the Array Controller (AC).

Above this layer is an optional scoreboard layer

15 which provides for several functional points where split offs can be taken from the mainstream.

Above the scoreboard there is an optional buffer, enabling more instructions to be passed out of the Thread Sequence Controller (TSC). This may be at the

20 penalty of exposing a bigger pipeline if the control mechanism loops back up to the thread manager without being able to do other work in the meantime.

Above the buffer is a decode entry point from the TSC. In essence, the generic controller can be

25 composed of as many segments in whatever configuration as required. There is also a signal port that each of the functional units can signal and, correspondingly, there is space in the instruction for signalling to be effected.

30 Different instantiations will involve the number of entry points, the pipes, and then fanning out via scoreboards in the pipes. The interactions between the elements in the pipes and the functions at the bottom of each pipe could potentially be different. They will

35 all conform to the same template of structure, so the

network processor will be a specific SIMD controller
made up of generic controller elements.

Referring now to the Array Controller (AC) in
Figure 5.1, ie the controller for the array of PEs,
this can be viewed as comprising two units. An
explanatory view of the AC is shown in Figure 5.4.

In Fig 5.4, a Load Store Unit (LSU) is provided,
along with the Array Sequencer (AS), in the AC. The
microcode for ALU (Figure 5.1) and its interaction with
the register file occur in the array sequencer. The
microcode relating to a pipeline PE coprocessor may be
"bolted on", for example a multiplier accumulate period
etc.

The BIO Controller (BIC) handles transfers to/from
PE memory to/from the outside world. It effectively
has a core coprocessor, called the transfer engine
(TE). Its job is simply to move data in and out of the
core. There is an equivalent controller/engine pair
for high speed streaming which is the DIO Controller
(DIC) and load/unload engine (LUE). The blocks to the
right of the Load Store basically look at the core as a
load, extract the data, process it and put the data
back into store. Everything that can be bolted onto
the core appears the same as far as the controllers are
concerned but the specific functionality differs. The
instruction set looks similar, the controller looks
similar, and the manner in which additional
functionalities are bolted on look similar.

Each Load/Store Unit (LSU) typically accepts at
least 32bit load/store requests and, in the case of a
load, returns data to the threads. The LSU uses the
fetch unit to send its requests onto the bus. The LSU
is likely to need a cache or buffer to make the thread
operations efficient, since the bus performs 256 bit
red/writes.

The fetch unit accepts requests from the cache in the form of an address and a tag. It performs a fetch on the bus and then returns the data with the tag. The fetch unit will accept fetch requests, one per cycle,

5   while it is not busy. It uses a control line to indicate its busy status to the cache and returns responses to the cache as the data arrives back from the bus. It indicates when responses return by raising a control line but it does not have to return responses

10   in order as the tag will inform the cache as to which was the relevant fetch.

The Transfer Engine (TE) and Load/Unload Engine (LUE) are similar except that they target different elements that were built into the PE Array. There are

15   two forms of IO, namely the BIO, which is the cell that connects to the PE, and DIO, which is a flavour of a portion of memory of a very simplified control structure. The DIO is a fairly simple IO structure that allows fast streaming and variable data size and works

20   on the large packets. It can be a master to the core, rather than the core driving it and it is thus possible for the data stream to control the core. As far as the network processor is concerned, it would be regarded as packet IO. It has the capacity to produce a much

25   larger bandwidth to it from the core.

Effectively, the DIO handles the chunks from the packets. As shown in Figure 5.5, these arrive at the LUE, which sits on the bus and knows how the memory is configured in the PE Array. This can be two portions

30   of memory with 128 bits of entry. The LUE knows that the memory has an "even" side and an "odd" side, and it knows the maximum size of a chunk. Data can thus be streamed in at twice the data-handling rate of the memories, provided buffering is handled properly.

35   Data is streamed in and out and the DIC is informed accordingly. The LUE copies it out into its

own PE area but it could alternatively be multibanked or dual ported and have the handshaking between the LUE and the load store or any other desired unit(s). Significantly, it is high bandwidth and puts data into PEs in order.

The data can be marked to indicate the size of the relevant chunk. Since the data visits the controller first and the controller has control of the addressing, it can actually say "We have only got 100 bytes to put in here, so I am only going to address up to this point, stop and move the whole machine down to the next PE." This permits a different size transfer to be performed "on the fly" as the data stream passes.

Whether or not all the chunk data is handled by the LUE in terms of loading it to the array of processing elements depends on a number of factors. The DIC is aware (to a certain extent) of the topology of what it is targeting. For example, it may need to know how many PEs are in the array and how many chunks have been sent to one block. If packets are not permitted to be spread over two blocks, then information needs to be made available for such situations as there being no more PEs or how much buffer is left for a packet. This can be done by flagging the stage reached in order to get back to the issue point (Figure 5.3) so that everything is aligned. From there, the core can be left to free run. It is deterministic. The memory employed for this may simply be a portion of memory with a very simple control unit handling a mark bit. The necessary handshaking need be implemented by no more than a couple of elements of memory so that, as data is put in, the PE is informed accordingly. On exit, a small pipeline for the mark bit is included in each PE. This memory can be enabled either on the first or the last mark bit so that the outgoing data can be enabled.

Block IO has a fixed number of quad bytes of data which are taken out with addressing data interleaved on an output bus and an input bus. The elements of the control structure are in each PE and all the

5 handshaking signals and control signals are handled by the Transfer Engine (TE). There can be up to about 15 modes of operation. The configuration makes it possible to perform such operations as write, modify, read, consolidated reads and writes, strided reads and

10 writes, and out of order returns. All of the "SMARTS" are decentralised and handled inside the TE. In this way, a core coprocessor transfer engine can be built that provides multiple profiles of data transfer. All of the mechanisms necessary to move data out from the

15 PE Array, control it and put it back are provided in this way.

The Processing Element is the basic item of the processor. In fundamental terms, it can be regarded as a simple ALU, capable of performing any function of two

20 inputs. It could, however, be a 16 bit ALU for an encryption engine, ie a small FFT engine. It is sequenced by the microcoded Array Controller. A unique part of the SIMD is that the Processing Element itself is able to determine whether or not to execute a stream

25 of data. This may be achieved by a simple enable stack, which is part of a status register. Alternatively, two or three streams of instructions may be selected. In order to accelerate processing speed, it is beneficial for the processing unit to have some

30 closely coupled memory. Prior art processors would typically include a hierarchical memory with a small block of very high speed memory plus a block of lower speed memory. In an embodiment of the present invention a multiple port register file is provided

35 instead. These can be large in area, so the number of permissible registers may need to be restricted. It

will generally be coupled with some memory. This
memory also helps because that is where most of the IO
is targeted, so any IO is targeted to the memory rather
than directly to a register file.

"Slices" of BIO, DIO and Any IO (AIO) structures
required, each of which is controlled by an equivalent
slice in the generic controllers, may couple to a bus
common to the Register File and memory. They can each
operate independently except when they have to access
common parts, such as the register file or the memory.
That is where the handshaking between those units will
occur and also with the array sequencer because it has
to "hijack" this bus so that it can pass data from, for
example, the DIO to memory, or even DIO to AIO. It
must then be able to grab hold of the bus. The array
sequencer in turn has to keep the ALU busy, therefore
the threads are separated between AIO operations and
the processing (ALU) operation.

The other controllers are those that target the
end points of those transfers. They are, for example,
embedded processors and the like and are effectively
the end point of one of the IO streams. The "any other
controller" mentioned previously could be a graphics
extension, for instance.

There are elements within the processor, such as
the BIO and DIO, that are unique. They provide
considerable functionality at the expense of very
little logic.

The individual aspects/blocks of the core will now
be considered in greater detail for a better
understanding of the operation, configuration and
architecture of the core. It will be remembered that
the block diagram of the network processor included a
plurality of first type processor elements (PE) and a
single second type PE.

The network processor itself consists of a number
of blocks, or "mini cores", each containing a
plurality of PEs. Typical values may be from 4 mini
cores, each containing 64 PEs, up to 8 blocks, each
5   containing 256 PEs or more. The modular nature of the
processor allows as many variations as are desired.
The schematic diagram of the overall processor, as
shown in Figure 5.1, indicates the mini core structure.
Each mini core consists of a number of processor
10   elements as above and contains a Direct IO (DIO), a
Block IO (BIO), an ALU, a Register File (RF) and a bank
of memory. A distribution block ensures that incoming
and outgoing data is properly distributed. A schematic
representation of this structure is shown in Figure
15   5.6.

    The DIO is schematically illustrated in Figure
5.7.1. Each DIO allows high speed streaming data input
and output to/from memory, in the sense that data flow
is addressless. (PEs cannot choose the address of
20   streaming data they receive.) The bank of memory used
in the DIO is separate from normal PE memory. Data can
therefore be streamed in/out without affecting the
normal PE operation (ie IO is a background task).
Transfers between PE memory and the DIO memory bank
25   occur in parallel for all PEs and are accomplished by
"hijacking" load/store cycles from the normal PE
compute task to implement a block copy.

    In the internal structure of the DIO module
illustrated schematically in Figure 5.7.1, the memory
30   bank for the DIO is the largest part of the module and
the word depth may be altered for different
applications. Two data ports are shown in Fig 5.7.1
but dual-port RAM or multiplexed single port RAM may be
used, the latter being preferable since only one port
35   can be used at any one time.

It is important to note that the RAM data port
to/from the LUE is wider than the LSU because accesses
driven by the LUE are effectively serial over each PE,
whereas the Load/Store port is driven in parallel over
each PE.

As for read and write mark bits, these are single
bit registers per PE that indicate which PE is taking
part in the serial data transfer to/from the LUE. Only
one PE in a block will have its read mark bit set at
any one time. Similarly for the write mark bit, as
only one PE is reading at any time and only one PE is
writing at any time. It is not permissible for reading
and writing to take place at the same time with respect
to the same PE, although two different PEs in an array
may be simultaneously reading and writing data.

The read and write mark bits are individually
shifted serially through each PE in turn until all PEs
in a block have been serviced. The shifting and
resetting of the mark bits is controlled by the LUE.
The LUE has separate control lines for shifting the
read and write mark bits. When performing a
simultaneous read and write on an array of PEs, the
write mark bit should always trail the read mark bit
(ie it should be marking an earlier PE), otherwise the
data that has not yet been read will be overwritten.

Each PE also contains a shift register that holds
the history of the read mark bit status. Typically,
this register is four bits long (although theoretically
it can be of any length) and provides a way for the LUE
to access two different PEs for reading at any one
time, namely the PE that is four PEs behind the "read
marked" PE. This allows the latency in the PE
distribution to be mostly hidden in situations where
some control data must first be read from a PE before
any other data can be read from the same PE.

Each PE in a SIMD processor may have none, one or more than one DIO but all PEs in a block should have the same number. Each DIO will normally require a dedicated Load/Unload Engine (LUE) as described later.

5   Figure 5.8 shows a case where one DIO is attached to each PE and one LUE is used to control the PE array.

An example of a PE is shown in Figure 5.9. The physical layout of the PE sub-panel effectively determines most of the schematic, verilog and physical

10   structure of the ALU, which should all be similar. Where the sub-panel consists of 4 PEs, the ALU will be constructed in fours, with each four sharing the majority of the control logic used to access the ALU. All the decoded control output signals are latched

15   before being used to drive the 4 ALUs.

The Transfer Engine (TE), a typical example of which is shown in Figure 5.7.2, is located between the system bus and the PE array and is controlled by the BIC "slice" of the Generic SIMD Controller (GSC).

20   The TE controls IO flow independently of the SIMD operation of the PE array; packs data into more efficient packets for transfer over the system bus; minimises the number of system bus packets on a sub-addressing boundary (consolidation); handles "out-of-

25   order returns"; gives a high flexibility for minimal area in the BIO; and puts the majority of variance in function in the semi-custom domain. The sub-address can be easily changed (4 or 16 quad payload) or multiple BIO/TE units can be instantiated in parallel.

30   At the Generic SIMD Controller (GSC), TE instructions will arrive in groups that cannot be broken up by thread switching. The GSC will command the LS to fill the BIO and will then issue the TE instruction and wait for completion before commanding

35   the BIO to save data if required. The unit can then signal, if required.

Figure 5.7.3 is a more detailed version of Figure 5.4. The GSC illustrated is composed of a number of controller "slices", each of which conforms to a specific structure. In particular, these slices are

5 the Array Controller (AC), the BIO Controller (BIC) and the DIO Controller (DIC). Each such controller slice receives a stream of instructions from the Thread Sequence Controller (TSC) and in turn controls a respective part of the SIMD PE Array. Figure 5.7.3

10 shows all of the required interfaces.

The Array Controller (AC), shown in Figure 5.7.4, is responsible for feeding microcode words to the SIMD core and to perform load/store operations between the PE memory and register file. In addition, it supports

15 the DIC and BIC by generating special load/store operations to service the DIO and BIO structures within the PE. The AC contains tables in RAM to hold the microcode and a microcode index table. It also performs all microcode expansion.

20 The BIO Controller (BIC) shown in Figure 5.7.5 is responsible for moving data between PE memory and the Block IO structures and for controlling the operation of the TE. There is an Inter-Controller interface between the BIC and AC to allow the BIC to generate the

25 appropriate load/store commands to service the BIO data and address registers.

Finally, the DIO Controller (DIC), a version of which is shown in Figure 5.7.6, is responsible for moving data between PE memory and the DIO memory bank

30 (block copy) and for controlling the operation of the LUE. There is an Inter-Controller interface between the DIC and AC to allow the DIC to instruct the AC to generate the load/store commands to implement block copy.

35 Detailed guidance as to the structure of the TE and of the BIO can be found in Figures 3.4.1 in the

"Generic Transfer Engine" engineering document annexed
to this specification.

Section 6

Lookup Table (LUT)

5   The Lookup Table (LUT) block does table lookups. A
table is a set of key-value pairs. The LUT treats both
the key and the value as arbitrary bit fields. Key
lengths can be up to 128 bits and values can be up to
41 bits. It does not assign any meaning to the bits of
10   the key or value; their interpretation is up to the
user. The LUT can be used for any table for which fast
lookups are required, including IPv4, IPv6 and MPLS
routing tables, flow tables and Access Control Lists
(ACLs). It supports multiple tables of the same or
15   different types. The LUT has some internal memory for
table storage, and it can also use external memory.
The LUT tables are stored in a special, compressed
format, called an LC-trie, to optimise lookup
performance. This format is fairly expensive to
20   generate, so the LUT is best used in applications for
which lookup speed is more important than update speed.

      The LUT lookup algorithm is based on a
level-compressed trie (LC-trie) data structure. The
25   data is organized as a set of linked nodes, organised
in a tree structure. Each trie node contains a
power-of-two number of entries. Each entry is empty,
points to another trie node, or contains the lookup
result. Entries that point to another trie node contain
30   a skip count and a skip value. Figure 6.1 illustrates
how this works. Some leading bits from the key are used
to index into the level 0 trie node. This is done by
adding the value of these bits to the base address of
the node. The entry in the level 0 node says to skip
35   some bits then use the next few bits to index into a
level 1 table. The entry in that level 1 node contains

the final value. In this example, two memory accesses were used to do the lookup, one each in trie levels 0 and 1. In practice, real tables contain many more nodes and levels than shown in this example. For example, an

5   IPv4 forwarding table with 100,000 entries might contain 6 levels and 200,000 nodes.

      The LC-trie algorithm is well known, and it has been studied in the literature. Its salient properties

10   in the presently-described example implementation are shown below (it will be appreciated that the figures given below are merely exemplary and are not intended to limit the description of the invention in any way:

15   1.   The size of a node entry is fixed at 6 bytes independent of the size of the key. This implies that the memory required for a given table does not depend on the key size. The format of a node entry is shown below:

20

field	bits	usage
bcnt	4	number of key bits used to index next node
scnt	4	number of key bits to skip
sbits	15	value to check against key when skipping
bindx	22	location of next node

25

30

      If bcnt is zero, the entry is empty, and does not

35   contain a value. If bcnt is non-zero, then the

entry point to another trie node. Values can
contain up to 41 bits.

2. The depth of a trie depends primarily on the
    number of entries in the table and the
    distribution of the keys. For a given table
    size, if the keys tend to vary mostly in
    their most significant bits, the depth of the
    trie will be smaller than if they tend to
    vary mostly in their least significant bits.
    A branch of the trie terminates in a value
    entry when the bits that were used to reach
    that entry determine a unique key. That is to
    say, when there do not exist two different
    keys with the same leading bits.

3. The nodes of a trie can contain many empty
    entries. Empty entries occur when not all
    possible values of the bit field used to
    index a node exist in the keys that are
    associated with that node. Simulations with
    IPv4 routing tables indicate that for such
    tables, about half the nodes are empty. Since
    the size of a node entry is 6 bytes, this
    implies that such tables will consume about
    12 bytes of memory per table entry.

One thing that is innovative about the presently-
described implementation of LC-tries is the inclusion
of a skip value field in each trie entry. During
lookups, the skip value field is compared to the
skipped key bits, and a lookup failure is signalled if
they do not match. In the known implementation of
LC-tries, skip values are not stored in the trie
entries, which gives rise to false hits in the table.
The possibility of false hits means that hits have to

be confirmed by performing an additional memory
reference to the full table. The present example
implementation eliminates the need for this extra
memory reference, at the expense of somewhat larger

5   entries.

Interface

The LUT presents three logical interfaces to
10   client blocks via a Virtual Component Interface (VCI)
compliant bus interface. See, for example, "VSI
Alliance Virtual Component Interface Standard (OCB
2,2,0), on-chip bus development working group, member
review version 24 August 2000", published by VSI
15   Alliance Inc.. All three logical interfaces share a
single VCI target interface. Alternatively, each
logical interface can be implemented by a unique VCI
target interface. There is a control interface for
initialization, configuration, and statistics
20   collection. There is a lookup interface for receiving
keys and sending results of lookups, and there is a
memory interface that makes the internal memory of the
LUT available as ordinary memory. All three interfaces
can be used concurrently, although naturally the
25   transactions will be serialised by the VCI bus
interface. It is possible to make use of the memory
interface while the LUT is busy doing lookups. Indeed,
this is how the tables in the LUT can be updated
without disrupting lookups in progress. The LUT can be
30   configured to use external memory visible on the bus
structure, in addition to, or instead of its own
internal memory. The LUT, although described with
reference to a network processor, can be used in other
contexts; it need not necessarily be part of a network
35   processor and any block with an VCI initiator interface
can make use of it. In particular, a control plane

processor could send lookup requests to the LUT via a Control Plane Interface if desired.

Various features of a particular example LUT will now be described:

1. Control. There are several internal registers that can be read or written. The control interface provides the following functions:

(a) Reset. All visible registers are set to default values, and any lookups in progress are aborted. The LUT powers up in this state.

(b) Setting the bit lengths of the key and value. There are internal registers for these lengths. These should not be written to while lookups are in progress. Valid values for key length are 1-128, and valid values for value length are 1-41.

(c) Memory size register. Reading from this register returns the size of the internal memory. Writing to this register has no effect.

(d) Setting and querying the parameters of external memory. At reset the LUT is configured to not use external memory.

(e) Reading and writing the count registers. There are registers for counting the number of lookups and the number of memory accesses. These can safely be read or written while lookups are in progress.

2.   Memory. The LUT internal memory is preferably
       organised as two equal size, independent
       banks. The size of these banks is a synthesis
       parameter. They are preferably organised as a
5   configurable number of entries with a width
       of 6 bytes. The maximum number of entries
       that can be configured for a bank is 131072,
       which implies a maximum total memory size of
       1.5 megabytes. Clients can use the LUT
10   internal memory in the same way as ordinary
       memory, bypassing the lookup state machines.
       The address for a memory access selects the
       entry for reading or writing. The data width
       for read and write using the memory interface
15   is 8 bytes, but the most significant bytes
       are read as zero.

3.   Lookups. Lookups are preferably performed
       using a split read transaction. Key values
20   are presented during a first phase of the
       operation, and the lookup result values are
       returned during a second phase of the
       operation. The following properties can be
       expected:

25

(a)   A lookup operation is supported for accepting
       multiple keys in a single bus operation.
       This is implemented as a client VCI write to
       the LUT with the data portion containing the
30   keys. The LUT responds by returning the data
       values associated with the keys. The LUT
       preferably has an input FIFO buffer with at
       least 128 slots, so it can accept at least
       that many keys without blocking the bus
35   structure.

(b)   Lookups that succeed return the value stored
in the table. Lookups that fail (i.e. the
key is not in the table) return a value
containing all 1 bits by default. More
precisely, the value returned is the value
found is the last inspected by the LUT lookup
state machine. It is feasible to construct
the tables in such a way that a lookup
failure returns additional information, for
example, the number of bits of the key that
matched in the table.

(c)   The LUT does not internally support longest
prefix matching, but that effect can still be
achieved by constructing the tables in the
proper manner. The idea is to split the
overlapping address ranges into disjoint
pieces.

(d)   Lookup values may not necessarily be returned
in the order of the keys. Each key is
accompanied by an 8 bit tag, and this tag is
returned with the value to assist client
functional blocks in coping with order
changes.

(e)   Multiple client functional blocks can submit
lookup requests simultaneously. If this
causes the input FIFO to fill up, the bus
between the requestor block and the LUT will
block temporarily. The LUT keeps track
internally of the source port of the
requestor for each lookup, so that the result
values can be sent to the correct place.

(f)   The contents of the memory being used by the
        LUT can be updated while lookups are in
        progress. The actual updates are done via the
        memory interface. There is a software
5   protocol, explained in a later section, that
        is used to guarantee table consistency.

        The LUT uses a number of lookup state machines
        (LSM) operating concurrently to perform lookups. Figure
10   6.2 shows the major internal blocks and their data
        connections. Incoming keys from the FB2 are held in an
        input FIFO buffer. The incoming keys are distributed to
        respective lookup state machines by a distributor
        block. Values coming from the state machines are merged
15   by a collector block and fed to an output FIFO buffer.
        From here the values are sent out on the bus structure.
        The input FIFO buffer entries each contain a key, a tag
        and a source port identifier. This FIFO buffer has at
        least 128 slots, so two client functional blocks can
20   each send 64 keys concurrently without blocking the bus
        structure.

        The distributor block watches the lookup state
        machines, and sends a key to any one that is available
25   to do a new lookup. A simulation of the LUT uses a
        priority encoder to choose the first ready state
        machine, but which one is chosen in any actual
        implementation is relatively unimportant.

30   In this example, the lookup state machines LSMs do
        the lookup using a fixed algorithm. They treat all
        keys as 128 bits and all values as 41 bits internally.
        Shorter keys are extended with zero least significant
        bits. Values are truncated to the configured value
35   size by discarding most significant bits. Memory read
        requests are sent to a memory arbiter block. The number

of memory requests needed to satisfy a given lookup is variable, which is why the LUT may return out-of-order results. These sizes have been chosen somewhat arbitrarily, and are therefore merely exemplary. It would be possible to extend the maximum key size to 256 bits at a fairly low cost. The main impact on the LUT would be an increase in the size of the input FIFO and LSMs. It would be possible to increase the maximum size of the result. The main impact would be that trie entries would be larger than 6 bytes, increasing the overall LUT memory required for a given size table.

The collector block serialises values from the lookup state machines into the output FIFO buffer. In one example, the first available value is decided using a priority encoder, but an implementation of the LUT could choose any available value.

The memory arbiter block forwards memory read requests from the lookup state machines to the appropriate memory block. This might be to an internal memory bank or an external memory via the bus structure. The LUT has a VCI initiator block for performing external memory reads. Whether or not a memory read request goes to external memory is determined by the external memory configuration registers.

The output FIFO buffer contains result values waiting to be sent to the requestor block. Each slot holds a value, a tag and a port identifier. If the LUT received more than one concurrent batch of keys from different blocks, the results are intermingled in the output FIFO buffer. Results are sent to the VCI interface using a chaining mode that groups consecutive results at the head of the output FIFO buffer going to

the same bus port, to improve efficiency on the bus.
This implies that all the results going to given block
may not be send together.

5   Results from running a simulation model of one
example of LUT via a test bench indicate that the LUT
can achieve a peak performance of about 300 million
lookups/second. These numbers are measured at the test
bench bus interface.

10

This level of performance is predicated on the LUT
internal memory system being able to sustain a memory
cycle rate of 800 million reads/second. This is be
achieved by using two banks of memory operating at 400
15   million reads/second with pipelining of reads. The
latency of the internal memory system needs to be on
the order of 4-8 cycles. Higher latencies can be
tolerated by increasing the number of lookup state
machines, but the practical limit is about 32 state
20   machines.

Usage

Preferably, the LUT state machine lookup algorithm
25   is fixed and fairly simple, to attain performance. The
way that the LUT achieves great flexibility in
applications is in the software that constructs the
LC-trie data structure. With this flexibility comes a
cost, of course. It is relatively expensive to generate
30   the trie structure. The idea for using the LUT is that
some general purpose processor, perhaps in the control
plane, preconstructs the trie data and places it in
memory that is visible on the bus structure, perhaps
the external SRAM block. An onboard embedded
35   processing unit (EPU) is notified that a table update
is ready and it does the actual update in the LUT

memory. Alternatively, any an external processor or
other controller can perform the table update process,
the EPU is merely one example of such a suitable unit.
When constructing trie structures you have to use the
right endianness. Here are several possible ways to use
the LUT:

1. Longest prefix matching. When constructing
    the trie from the routing table, one can look
    for overlapping ranges and split them. This
    preprocessing step is not very expensive and
    does not significantly increase the trie size
    for typical IPv4 routing tables.

2. Multiple concurrent tables. This can be
    achieved by prepending a small table
    identifier to the key. With eight tables, the
    cost for this would be three bits per key.

3. Returning the number of matching bits. The
    lookup engine returns whatever bits it finds
    in the last trie entry it fetched. Further,
    on a lookup failure that entry is uniquely
    determined by the lookup algorithm; it is the
    entry that would have contained the value had
    the missing key been present. The program
    that generates the trie structure could fill
    in all empty trie entries with the number of
    matching bits required to reach that trie
    entry. These return values could be flagged
    some way to distinguish them from lookup
    table hits by the generator program. Then
    the LUT would return the number of matching
    bits on a lookup failure.

4. Concurrent lookups and updates. The best that
    can be done is to make sure the tables are in
    a consistent state at all times. Then lookups
    in progress will find either a value from the
    old version of the table or a value from the
    new version of the table. An EPU performing
    an update achieves this by first placing the
    new level 1--n nodes in the LUT memory, then
    overwriting the level 0 node entry that
    points to the new nodes.

5. Very large results. If a value for a given
    key needs to be more than 41 bits, an
    auxiliary table can be placed in the LUT
    memory - actually any available memory - and
    an index into the auxiliary table placed in
    the LUT value. The auxiliary table would
    then be read using normal memory indexing.

It will be appreciated that a lookup table structure and method of operation in accordance with that described above can provide many advantages over previously-considered techniques.

5

According to an embodiment of another aspect of the present invention, there is provided a lookup table which comprises a plurality of value storage levels, the storage levels being accessible via input key

10   values.

According to another aspect of the present invention, there is provided a method of performing LC trie lookups to produce table lookup results.

Section 7 — SEMAPHORE UNIT

The semaphore unit is connected to the bus along with all of the other modules of the network processor. The purpose of the semaphore unit is to coordinate communication between the various modules that are connected to the bus. This form of control is required in order to prevent, for example, two separate cores of the SIMD array trying to access the same data simultaneously.

The provision of a separate semaphore unit enables synchronisation functions to be removed from each of the individual modules that constitute the network processor.

This has the advantages of simplifying the hardware in each of the network processor's modules, and provides greater flexibility. The semaphore unit provides a generic control mechanism that can be used by any module under software control.

As mentioned above, the semaphore unit is connected to the bus. Each semaphore it contains is memory mapped into the address space of the semaphore unit. The number of semaphores provided by a module, and the number of modules attached to the bus can be tuned to any particular application.

A semaphore is an entity that can be both signalled and waited on. If another module on the bus issues a wait request on a semaphore, it will not be satisfied until the semaphore has been signalled. Any module may signal the semaphore. The order of arrival of signals and waits is not fixed, however, each wait can only be satisfied by one signal.

According to one embodiment, each semaphore carries out a count of each signal received when there is not a currently pending wait, and also queues a list of pending waits if no signals are available to satisfy them.

According to another embodiment, an item of data is attached to each signal, which is then returned to the wait that is matched with it. In this embodiment, the signals that are received are queued in addition to being counted.

Preferably, a semaphore state is accessible via the bus, thereby allowing the unit's context to be saved/restored or otherwise modified.

Thus, the semaphore unit described above provides the functionality that enables more than one module to signal a semaphore and more than one module to wait on a semaphore, with the order of the signals and the order of the waits being matched together.

To summarise, according to the simplest exemplary embodiment of the semaphore unit, each semaphore occupies X bytes of the address space. Each write to that address is recognised as a signal. Each read is recognised as a wait. The read does not return data until a signal has been received. The value written is discarded and the value returned is undefined. The maximum number of pending signals or waits is one, i.e. there is no counting or queuing.

According to another embodiment, the semaphore unit additionally allows the value contained in the signalling write to be returned to the waiting read.

According to another embodiment, the semaphore unit additionally allows multiple pending signals and waits. This is achieved by counting the signals and queuing the waits.

Thus, the semaphore unit provides the flexibility of enabling the values written to be ignored if desired, or, alternatively, to be added to the semaphore unit count value.

Similarly, the semaphore unit has the option of treating the read values as being undefined, or, alternatively, they can be defined to contain the current value of the semaphore counter.

According to yet a further embodiment, the semaphore unit allows the value contained in the signalling write to be returned in the read of the wait to which it is matched. This replaces the counter increment/examine options given in the previously mentioned embodiment. According to this embodiment, a queue is maintained of the pending signals instead of just counting.

The functionality of the semaphore unit described above is made possible by the nature of the split transaction of the bus, i.e. that reads are two separate transactions, a request and a response.

The semaphore unit may be implemented, for example, by implementing a complete semaphore device and replicating for each semaphore or by building one controller and providing it with access to a block of memory in which it stores the state of each of the semaphores it projects to the outside world.

The embodiments described above provide a separate
semaphore unit which is connected to the bus of the
network processor, and provides generic control over
each of the other modules connected to the bus.

5

The semaphore unit appears as just another module on the
bus, but in the form of "smart"memory. Writing to the
memory constitutes signalling a semaphore, and reading
from the memory constitutes waiting on a semaphore, since
10   the read value will not be returned until the semaphore
has been signalled.

The semaphore unit described above has the advantage
that no additional special logic is needed in any of
15   the network processor's other modules to support the
semaphore unit. In other words, modules such as the
SIMD array, thread manager or thread sequence
controller, all have the ability to perform load and
store operations, thereby enabling them to access the
20   semaphore unit without requiring any additional
hardware.

These and further features of various embodiments of
various aspects of the present invention can be found in
25   the attached annexes.

SECTION 8

DATA PACKET PROCESSING

5   Section 8 will describe the processing of data
packets that are input to a network processor. Figure
8.1 is a schematic view showing a data packet flow
being input to a network processor. The data packet
flow is received by a network input processor NIP. The
10   NIP distributes the incoming data packet flow amongst
processing elements contained in one or more processing
cores. In a preferred example, each processing core is
a SIMD processing core having multiple processing
elements (PE), and the NIP divides the data packet flow
15   into respective data portions (or "chunks") for the
PEs. The processing core can be made up of a single
array of PEs or can comprise a plurality of "mini-
cores" which have a smaller number of PEs. Using a
group of smaller mini-cores has several advantages.
20   The size of the data portion or chunk supplied to
each PE by the NIP is variable and is preferably set by
the software controlling the processing core. In one
preferred example, the portion size is larger than the
header size of an incoming data packet.
25   A typical data packet is illustrated in Figure
8.3, and comprises a data packet header (DP header) and
payload (DP payload). The exact structure of the
packet is dependent upon the data packet transfer
protocol in use by the system concerned, and is not of
30   importance for the purposes of explaining this aspect
of the present invention.
         The NIP divides the data packet into a number of
discrete data portions for supply to individual PEs.
Figure 8.4 illustrates data portions for supply to

three PEs PE1, PE2 and PE3. Each data portion comprises a control word CW with an appropriate portion of the split data packet. In the example illustrated in Figure 8.4, PE1 receives a control word CW1 followed by the DP header and a portion of the DP payload. PE2 receives a control word CW2 followed by a further portion of the DP payload, and PE3 receives a control word 3 CW3 with the remainder of the DP payload.

The NIP supplies the divided data packets to the PEs such that each new data packet begins at a new PE. This may mean that some areas of PE memory are not filled by respective portions of a divided data packet. The NIP supplies an integer number of divided data packets to each processing core or mini-core. In other examples, it may be possible for the NIP to spread a data packet across two or more processing cores.
Each PE receives a data portion and stores this portion in PE memory. The locations of the storage of the data in PE memory can be arbitrarily defined. In one preferred example, the control word is always stored in the same location in PE memory. This allows control of the PE to be simplified, since each PE will always have CW data stored in the same part of memory. The storage of the data part of the data packet portion (chunk) can be more flexible. For example, referring to Figure 8.5, the start Si and end Ei addresses in memory can be defined, such that data from the data portion for that PE is stored in an appropriate part of the PE memory.

When all of the available PEs in the core (or in the mini-core) have been supplied with respective data portions, those data portions can be processed. The processing of the packet data is illustrated generally in Figure 8.2. Initially, those PEs which have data to be processed are identified. It is possible that some

PEs remain unused, or have existing data that does not require further processing, but rather requires some other action, such as an output process, to be carried out.

5   Not all of the PEs which hold data portions will need to be instructed to process those data portions. For example, some PEs may only be holding data portions relating to the payload of a data packet, and so for data packet routing purposes will not be required to

10   operate on the data portions. For routing purposes, only those PEs holding data portions relating to data packet headers will need to process the held data portions.

The processing is carried out in line with single

15   instruction multiple data (SIMD) techniques; that is, all PEs in a processing core (or mini-core) carry out the same instruction at the same time, but operate on respective data portions.

The first processing task that each PE must

20   perform is to classify to which type of data packet the stored data portion concerned belongs. The actual technique used to classify the data portion is not important for this description, but must simply be able to indicate to the PE the type of data portion stored

25   in PE memory. For example, different communications protocols will use different data packet formats that must be processed in particular different ways. The classification of the data portion is intended to identify which processing steps are appropriate to the

30   packet concerned.

The PE will then extract a number of data bits from the data portion in order to build a key to a lookup table. In a typical example for handling routing of the data packet, the extracted bits relate

to the destination address for the data packet (for example in a level 3 processing case). The extracted bits could also include data relating to other information such as destination and source addresses, protocol type, or quality of service requirements, for example.

The extracted bits are used to look up values in the lookup table. For the case when the bits represent the destination address, the lookup table result will supply information relating to the switch settings required to route the data packet through the switch fabric of the router. Since the table lookup process will take a relatively large amount of time, the processing element can also undertake processing which is independent of the look up process in parallel with the look up process. While waiting for the result of the table lookup, the PE can for example perform tasks such as checking for errors in the packet or can decrement a time to live or other parameter of the data packet or data portion. The total compute time available in the processing element may amount to a thousand cycles, the table lookup may amount to around two or three hundred cycles. It is therefore important to be able to perform other processing tasks in parallel with the look up. It is also possible that the table independent processing could calculate results which will be dependent upon the result of the table lookup in a predictive manner. The use of the result would then depend upon the actual result of the look up process.

When the table lookup result is received by the PE, and the lookup independent processing is completed, the table lookup result is checked. Initially it is checked to see if the result is useful, because there

may be one of various error conditions that could exist in the table. The result of the table independent processing and the result of the table lookup will determine whether modifications need to be made to the data portion being processed by the PE concerned. For example, the table lookup could indicate that the data portion being processed belongs to a "tunnelled" data packet.

A tunnelled data packet is a packet which contains a further data packet within it. By way of example, tunnelling occurs when a data packet travels from A to B via an intermediate node C. The data packet issued by node A contains destination information pointing to node C. When the data packet arrives at node C and the header information is examined, it is found that the data packet payload of the initial packet in fact contains further header information and the actual payload. The initial node A header is then removed to reveal the actual destination for the packet. The packet can then be routed from node C to node B.

Once the packet has been modified, as required, it can then be sent to its destination. The PE may choose to send the data packet directly to the switch fabric for routing out of the router to the destination. This will probably be the most common result for the processing of the data packet. In the network processor, a network output processor NOP reconstitutes the full data packet from the data portions spread across the PEs of the core (or mini-core) concerned.

The reconstituted data packet that is transmitted to the switch fabric includes additional information regarding the switch settings required to output the data packet in the correct route. The switch information is included in a leading switch control

word.

In addition or alternatively, the data packet may be sent to the control plane of the router. The primary reason for sending a data packet to the control plane is that the PE concerned does not know how to process the data. This may be the case because the PE does not have sufficient processing time available to determine the particular requirements for the data packet concerned. In addition, there may also be data packets which are simply destined for the router, for example containing routing protocol information, statistical requests etc.. All those sorts of requests are usually handled by the control plane and not by the network processor.

It may be that the data packet contains errors or requires too much processing time, so that the data packet is simply dropped.

The fourth option is to re-circulate the packet. Recirculation simply means making a packet available again so that further processing can be carried out on the packet, or data portion. For example, using the tunnelling example of above, it can be seen that once the original data packet is considered to be a tunnelled packet, it will be necessary to process further the information contained in the original data packet to ascertain the actual destination for the data packet concerned. When the outer header information is removed from the tunnelled data packet, the real header information must be used to route the now de-tunnelled data packet to its correct destination.

Another possibility is that the data packet must be transmitted to multiple locations, using a multi-cast technique. In that case, when the packet is output from the NOP it is supplied to the switch

fabric, as well as being resupplied to the NIP for re-circulation.

Re-circulation can be implemented in several different ways. In a first method, data portions can simply be retained and are marked as such so that the data is not extracted from the PE. When new data portions arrive into the core, PEs that already contain data will not be offered new data. This has the advantage that the hardware remains relatively simple and delays can be kept to a minimum.

A second solution operates to re-circulate data portions (chunks) within a core (or mini-core), i.e. within the group of processing elements themselves without the need for reconstruction of the complete data packet. Data portions are simply transferred from one PE to another within the same core (or mini-core).

A third solution implements the further processing of data portions by sending all of the data portions from the core to the NOP for reconstruction into the complete data packet. Following the reconstitution of data packets, those data packets can be re-circulated by inputting the reconstructed data packet to the NIP.

A fourth solution is to reconstruct the data packet within the mini-core, and recirculate it within the mini-core. The mini-core will then have to split the reconstructed data packet into data portions for supply to the Pes.

As described above, when data is loaded into a memory of the PE, the area of the PE memory allocated for the incoming data portion can be defined arbitrarily within the total PE memory. Figure 8.5 illustrates the definition of a start point Si and an end point Ei. The control word associated with each data portion is preferably always stored at the same

address in PE memory.

When outputting data from the PE memory, the start
and end addresses of that memory for output can be
defined by the PE, as illustrated in Figures 8.6, 8.7

5   and 8.8. In Figure 8.6 the output start address So is
defined ahead of the input start address Si, which
effectively means that the PE can pre-pend data to the
data portion that it has been processing. This could
be necessary when routing information is to be added to

10   the data portion. Figure 8.7 illustrates a situation
when the output end address Eo is after the input end
address Ei, so that the PE can effectively append data
to the data portion. Figure 8.8 shows a situation when
the output start address So is positioned so as to

15   effectively delete part of the data portion on which
the PE has been operating. For example, the data
packet header can be removed in the case of a
tunnelling data packet.

The data packet header information is preferably

20   processed by a single PE. However, when the header
information is larger than amount of storage available
in the PE memory, it will be necessary for multiple PEs
to process the header information. This can be an
advantage in the situation where the header indicates

25   that multiple look up are required. An individual PE
can only perform one look up at a time, and so having a
single PE perform multiple look ups will take an
undesirable length of time. Using multiple PEs to
perform multiple look ups means that these operations

30   can be carried out in parallel, thereby reducing the
overall time taken to perform the desired number of
look ups. Even when the header information is of a
size that means it can be stored in a single PE memory,
the PE concerned could enlist other PEs to perform

specific table look ups.

According to this aspect of the present invention
there is provided a method of processing a data packet
stream in which method data packets, or portions

5   thereof, are supplied to a plurality of processing
elements for processing in a single instruction
multiple data fashion in which the processing elements
process respective data portions in accordance with a
common instruction. Such a method typically ascertains

10   routing information for the data packets concerned.

In accordance with another method embodying as
aspect of the present invention, incoming data packets
are divided into data portions for supply to respective
processing elements, the data portions are recombined

15   to form a reconstructed data packet following
processing of the data portions by the processing
elements.

These and further features of this aspect of the
present invention can be found in the attached annexes.
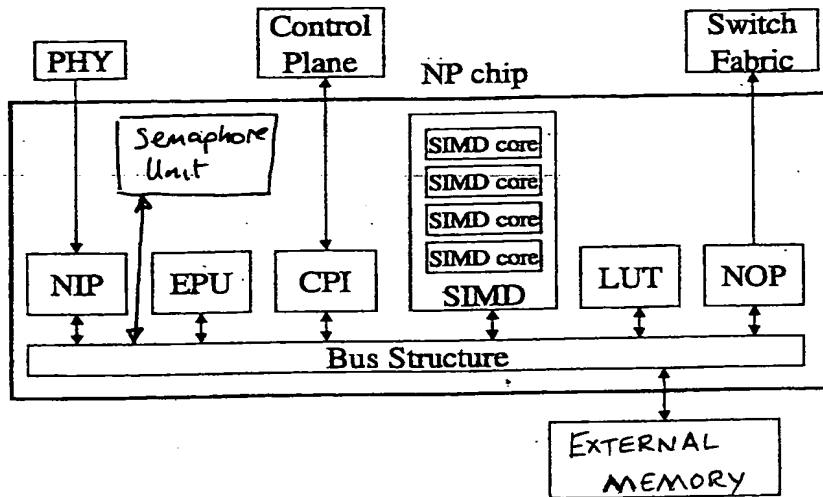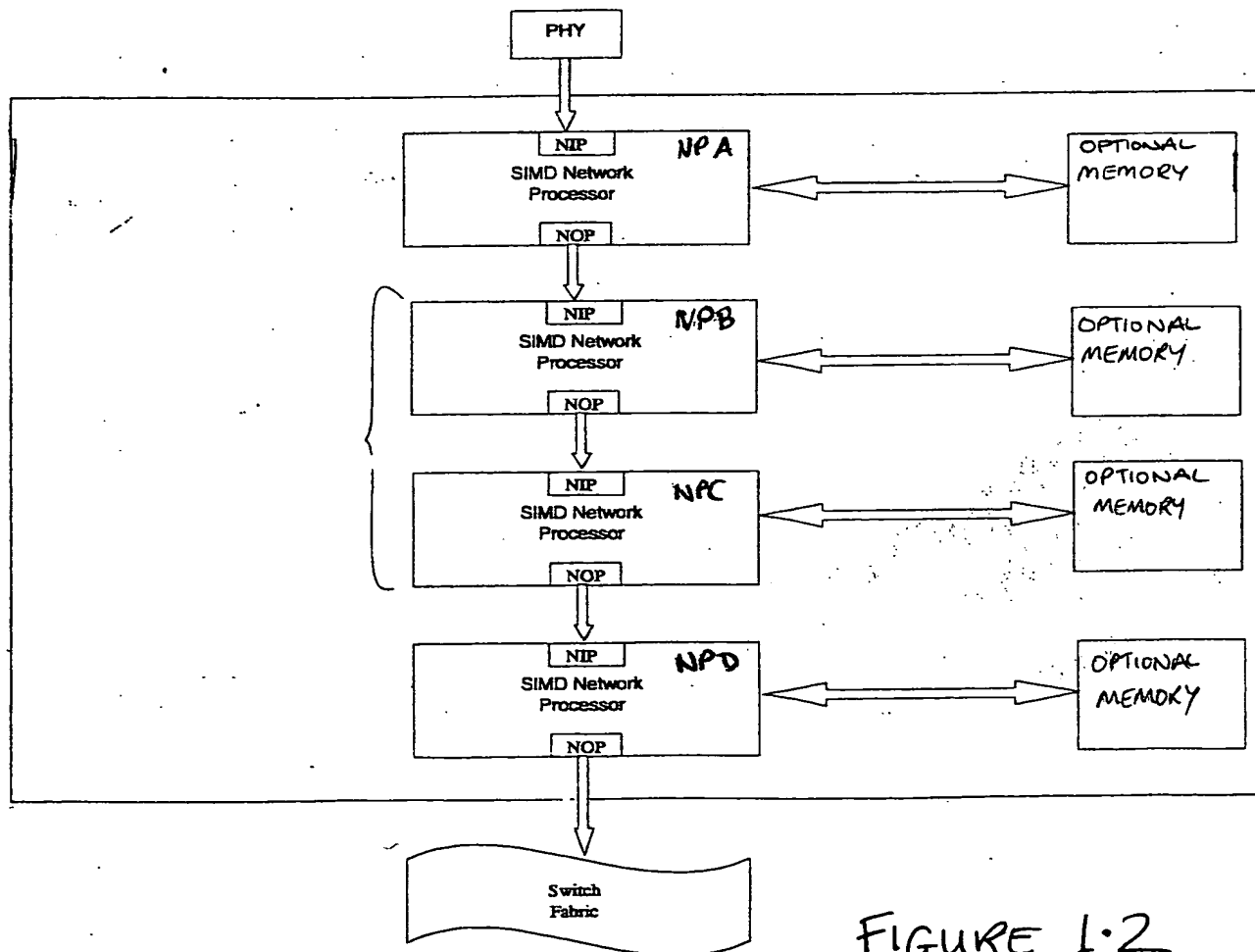
20

FIGURE 1·1

FIGURE 1·2

FIGURE 1·3

REGISTERS

Length and type

Data

Control

GIO Input Interface

Packet Buffer

De-chunked buffer

FIFO

NIP to Bus Input interface

NIP State Machine

FIFO

NIP to Bus Output Interface

FIG 2.1

Packet Header	Packet Payload (PL)

FIG 2.2

CH	Packet Header	PL		CH	PL		CH	PL		CH	PL
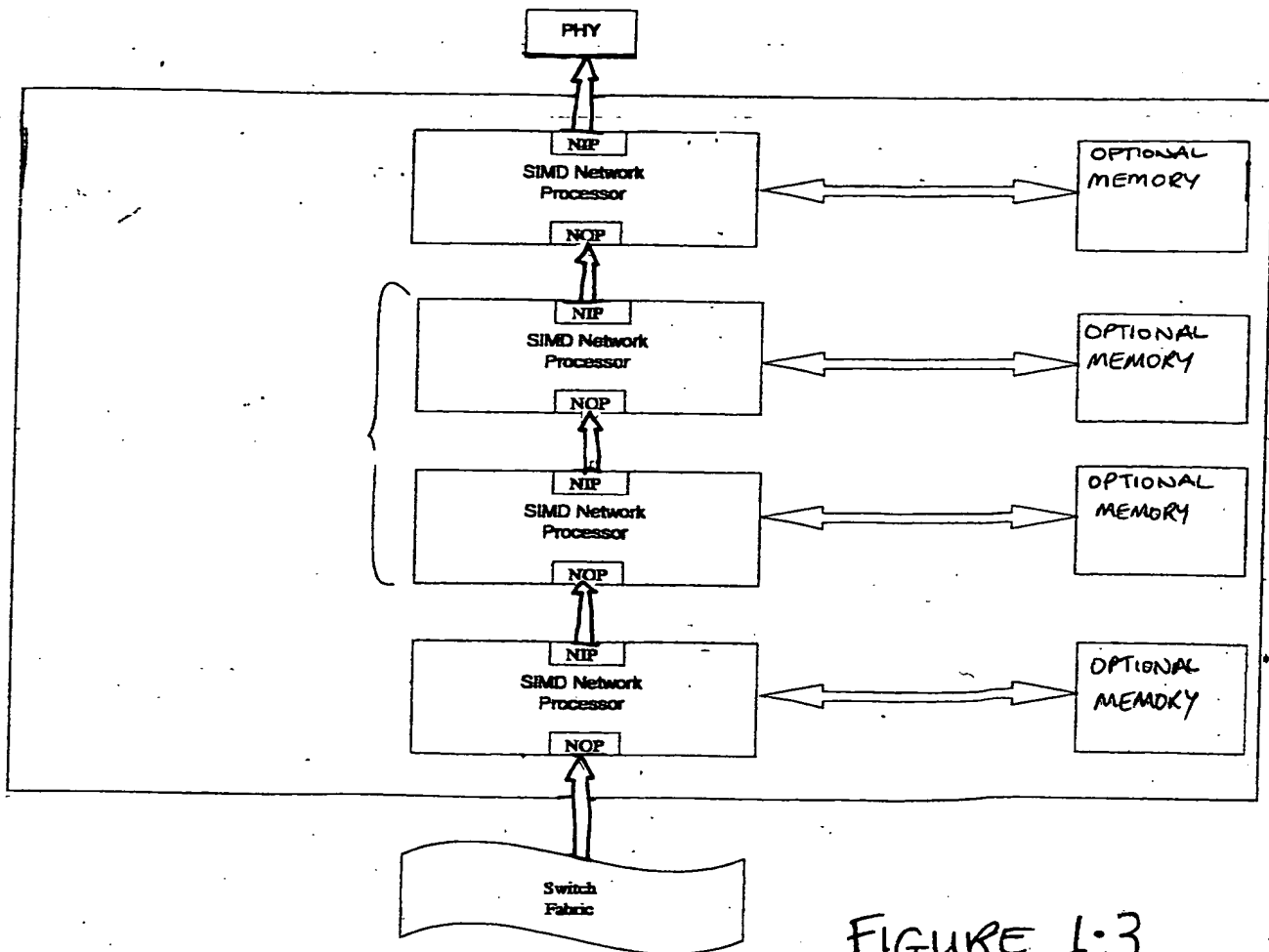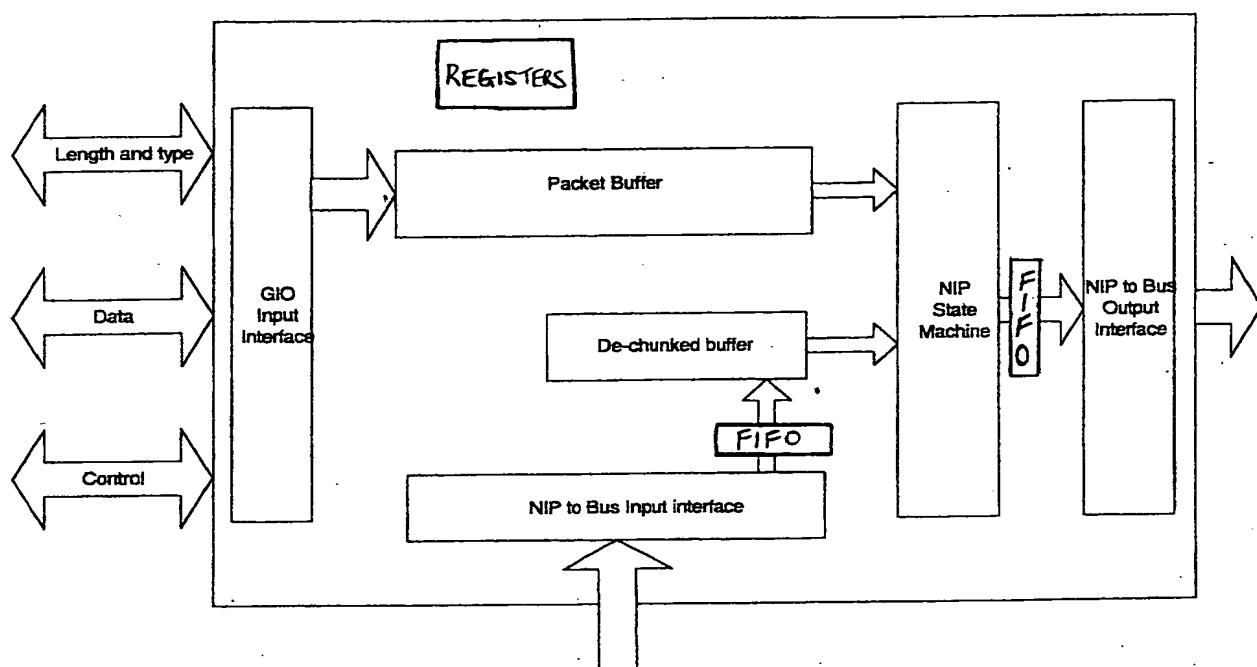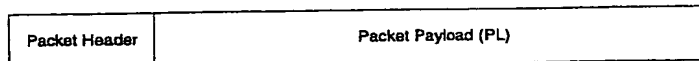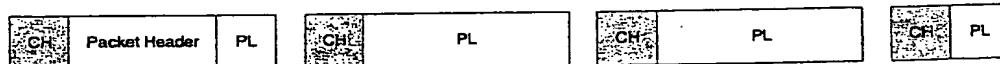
FIG 2.3

THIS PAGE BLANK (USPTO)

63	48	47	32	31	24 23 22 21 20 19 18 17						11	10
Load Address		Layer 3 Protocol Type		Reserved	D	P	E	R	F	C	Chunk Identifier	Length of Chunk

Load Address	The address where in DIO memory bank to load the Chunk
Layer 3 Protocol Type	The Data Link Layer protocol field as extracted by layer two
E	Error - discard
P	Send to control plane
R	Recycled chunk
F	Final chunk
C	Chunked or not
Chunk Identifier	Sequence number of the chunk
Length of Chunk	Size of the chunk excluding the prepended word
D	Downstream

FIG 2.4

REGISTERS

NOP to Bus Input Interface	F I F O X
	F I F O
	F I F O X

NOP State Machine

Packet Buffer

Recycling and exception buffer

FIFO

NOP to Bus Output interface

GIO Output Interface

Length and type

Data

Control

FIG 3.1

| Ctl | PreH | Packet Header | PL | | CH | PL | | CH | PL | | CH | PL |

FIG 3.2

| PreH | Packet Header | Payload (PL) |

FIG 3.3

figure 4.1

FIGURE 4.2

figure 4.3

Packet (arbitary length)   **Flit**

Byte Masks (16bits)

Data (128bits)

Head (1bit)
Tail (1bit)
Addr Info (16bits)
control, mem addr etc

Direction of data Flow

FIGURE 44

Block 1	Block 2	Block 3	Block 4

Packet B header flit   Packet A tail flit   Packet A header flit

FIGURE
4.5

146 bit registers

FIGURE   4.6

← incoming flit(s) blocked

146 bit register	146 bit register

to / from module
B

receive buffer
(consume
controller)

inject buffer

(inject controller)

module injects own flit(s)

FIGURE 4.7

146 bit register	146 bit register

to / from module
B

receive buffer

inject buffer

module receives right-going
flits while injecting its own
right-going flits

FIGURE 4.8

Linked, sending data

Linked, not sending data

Bus Lane

FIGURE 4.9

FIGURE 4.10

FIGURE 4.11

Figure 5.1

15/29

Fig 5.2

Faux Bus

Report/AEO

Memory Banks

Thread Processor

Scheduler

Instruction Caches/Data

TMBus

Semaphore Unit

Fetch Unit

FBI

Signal Sources

Thread Sequence Controller

THIS PAGE BLANK (USPTO)

Fig 5.3

TSC

pipes

decode entry point

buffer

scoreboard

ƒ

issue point

signal port

P.E ARRAY

(Ac)
Array Controller

TSC

AS

LOAD/
STORE

μ CODE

PE Array

Fig 5.4

TSC

TSC

BIC

DIC

TE

LUE

BIO

DIO

per PE

Array of PEs

Fig 5.5

NETWORK
PROCESSOR
(bus)

LUE

128 bit

| 0 | 2 | 4 | ... |

| 1 | 3 | 5 | ... |

128 bit

Fig 5.6

multiple register file

| ALU | RF | MEM |

AS

LS   Bioc   Dioc   Aioc

| ALU | RF | | Bio | Dio | Aio | Bus | PE MEM. |

PE COPROCESSOR

Fig 5.7.1

Internal structure of Direct IO module

Fig 5.7. 2

System Bus

SC
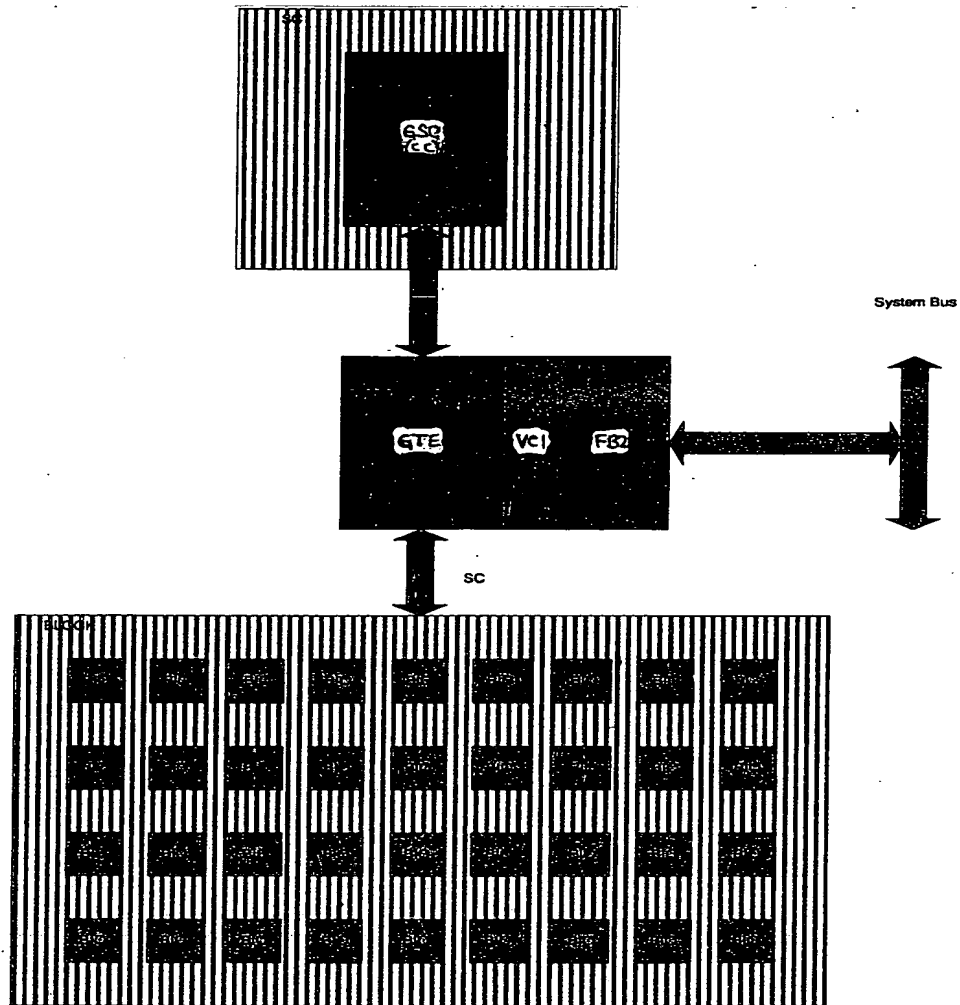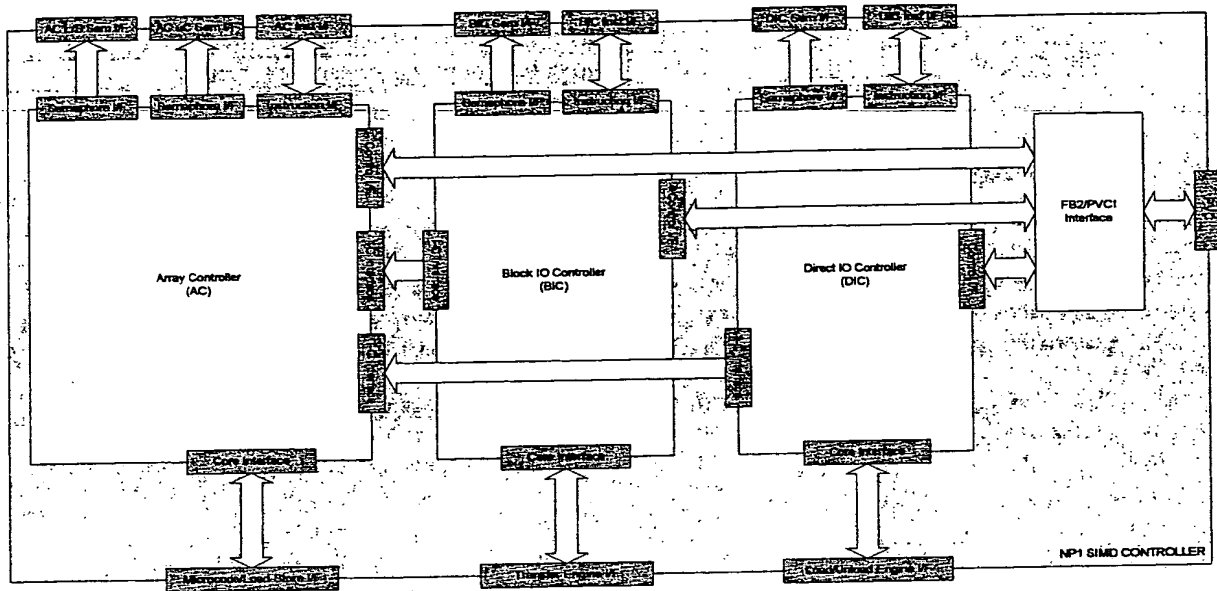
Fig 5.7.3

NP1 SIMD Controller Top Level

Fig 5.7.4

Fig 5.7.5

```
┌─────────────────────────────┐
│   BIO  INSTRUCTION  I/F      │
└─────────────────────────────┘
              │
              ▼
     ┌──────────────────────┐
     │   TSC  INTERFACE     │
     │   (siding buffer)    │
     └──────────────────────┘
┌──────────────┐      │
│ BIO Control  │──► TO ALL SUB-BLOCKS ──►
│    I/F       │      │
└──────────────┘      ▼
     ┌──────────────────────┐
     │  PRIORITY  QUEUING   │
     │   (Hi  Med  Lo)      │
     └──────────────────────┘
              │
              ▼
┌──────────┐  ┌──────────────────────┐  ┌──────────────────┐
│ BIC      │◄─┤                      ├─►│ BIC INTER-       │
│ SEMAPHORE│  │   EXECUTION  UNIT    │  │ CONTROLLER  I/F  │
│   I/F    │  │                      │  │                  │
└──────────┘  └──────────────────────┘  └──────────────────┘
              │
              ▼
     ┌──────────────────────┐
     │ TRANSFER  ENGINE I/F │
     └──────────────────────┘
```
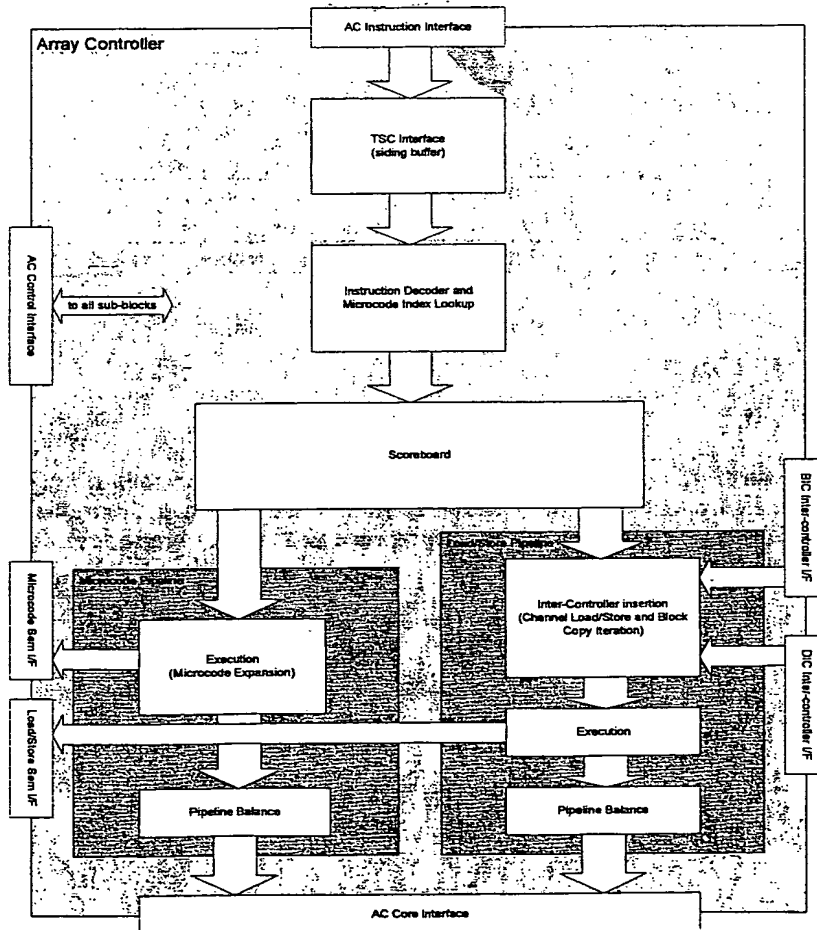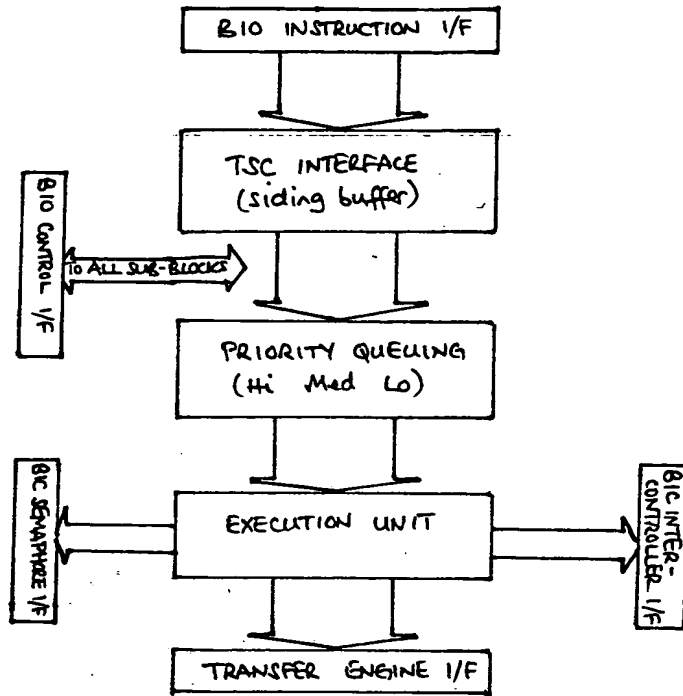
Fig 5.7.6
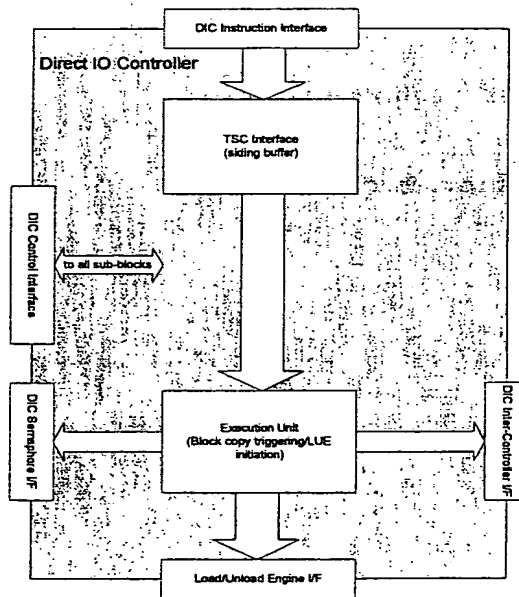
Direct IO Controller Structure
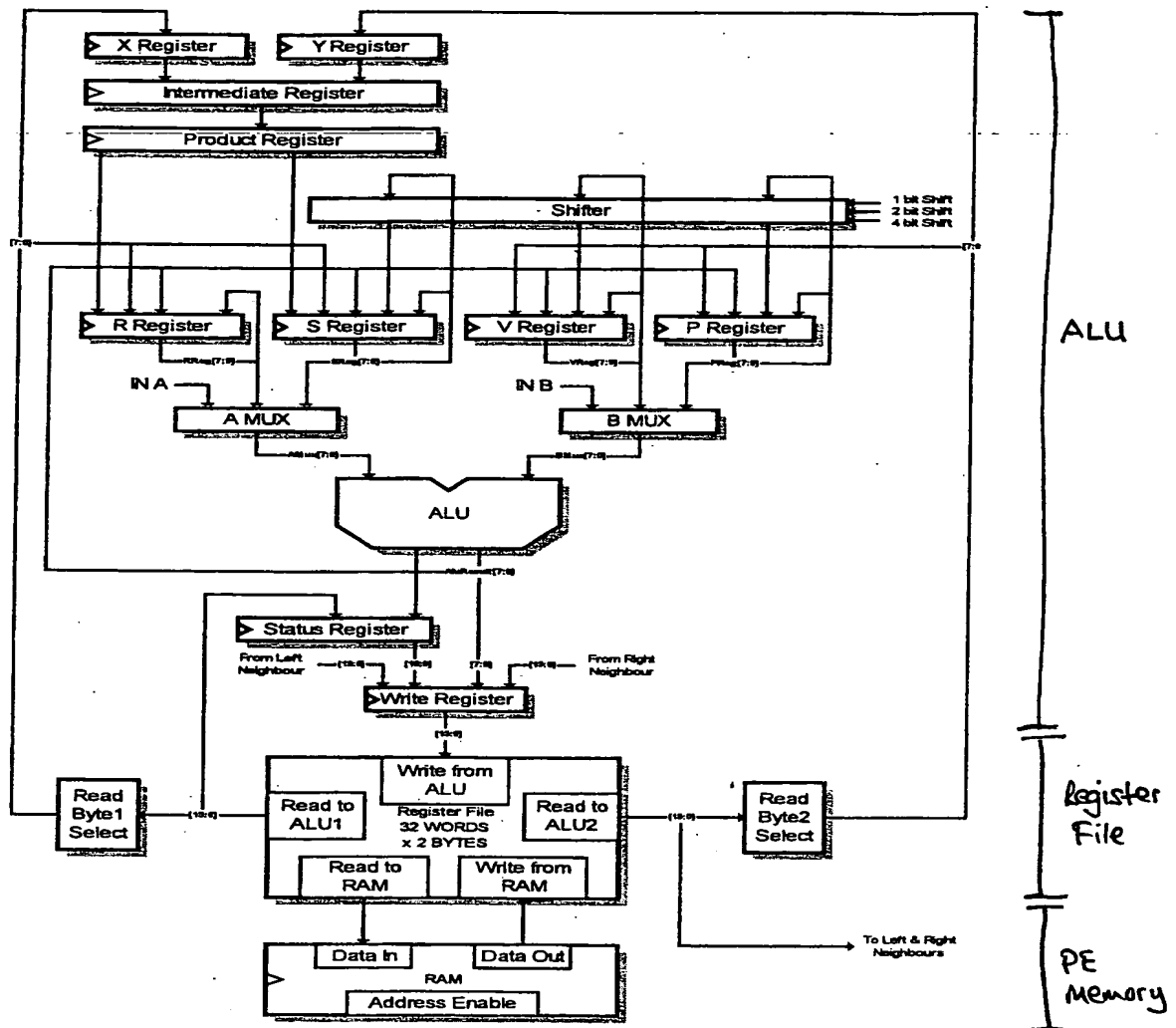
Direct IO in PE context

Fig 5.8

Fig 5.9

FIGURE 6.1

FIGURE 6.2

DATA PACKET
FLOW

NIP

CORE
[PE] [PE]

NOP

Figure 8.1

ACTIVE PES?

CLASSIFY
PACKET

TABLE
LOOKUP

TABLE
INDEPENDENT
PROCESSING

CHECK LOOKUP
RESULT

MODIFY
PACKET

SEND
TO:

SWITCH FABRIC

CONTROL PLANE

RECIRCULATE

DROP

Figure 8.2

DP HEADER	DP PAYLOAD

Figure 8.3

PE1

C W$_1$	DP HEADER	DP PAYLOAD

PE2

C W$_2$	DP PAYLOAD

PE3

C W$_3$	DP PAYLOAD	

Figure 8.4

PE1

S_i e_i

C W$_1$		DP HEADER	DP PAYLOAD	

PE2

s_i e_i

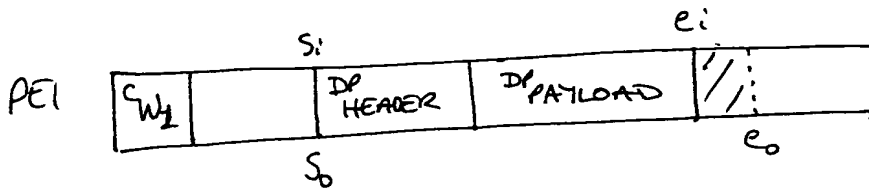| C W$_2$	DP PAYLOAD	

PE3

s_i e_i

C W$_3$	DP PAYLOAD	
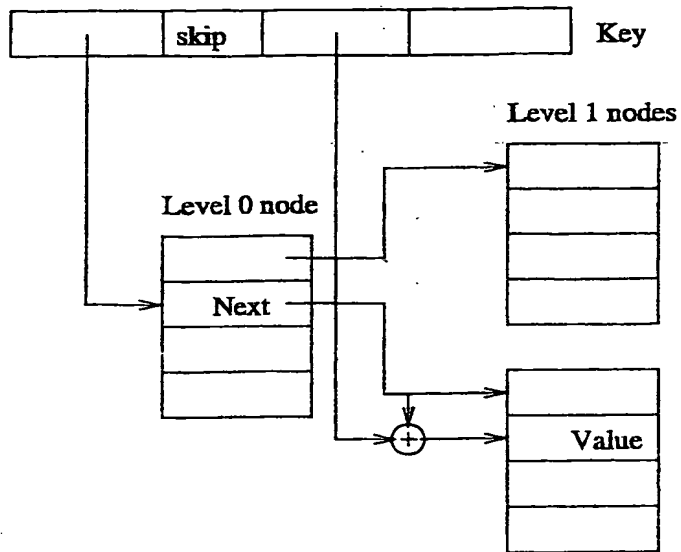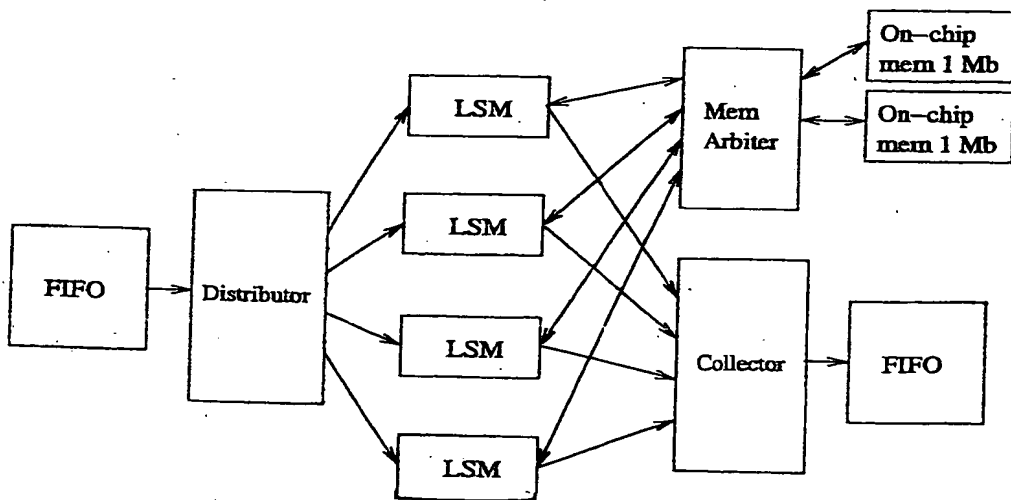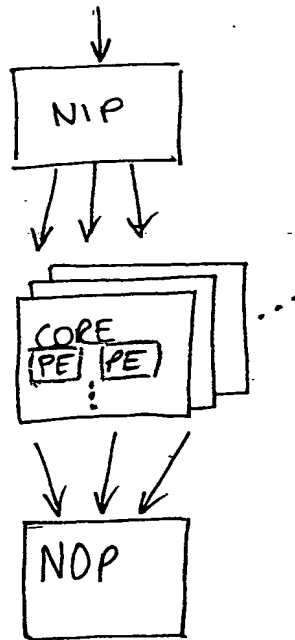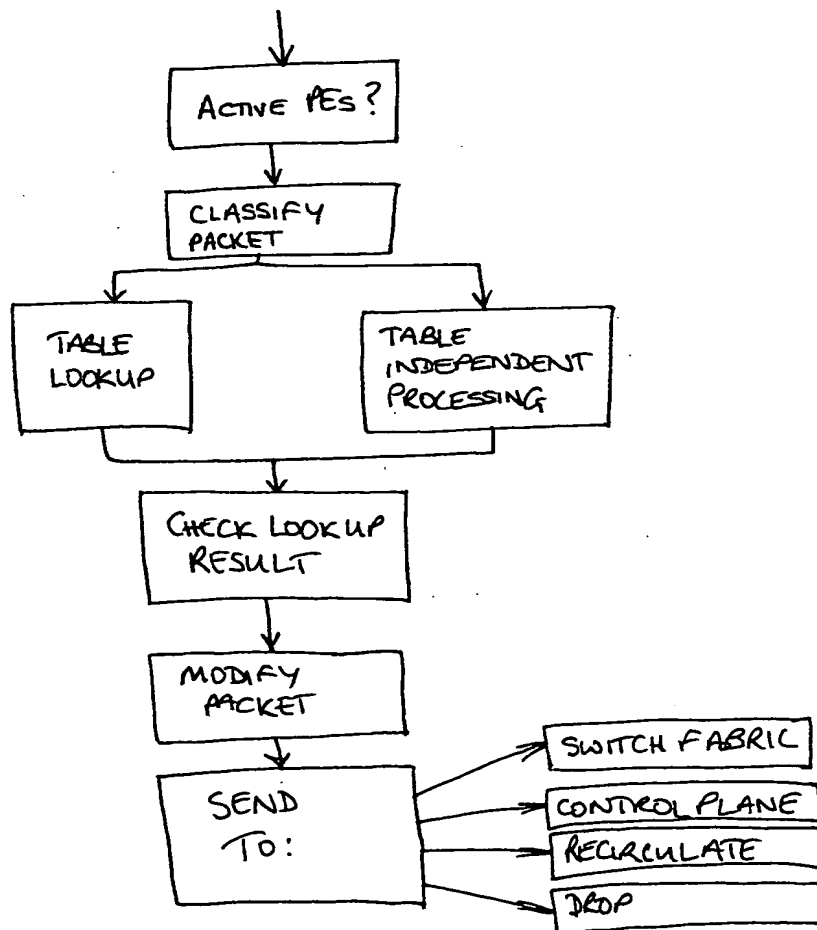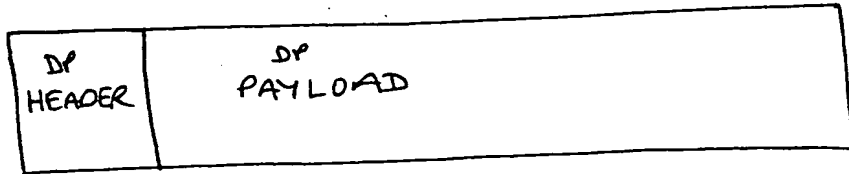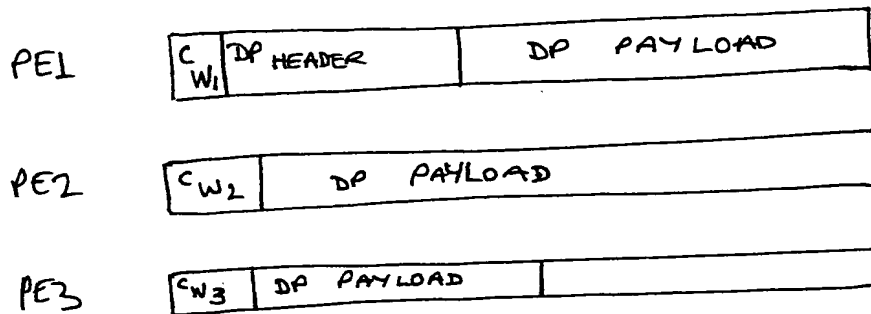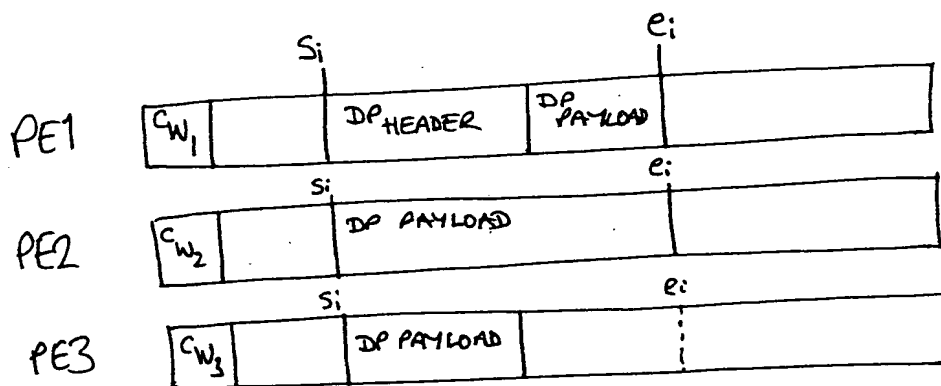
Figure 8.5

THIS PAGE BLANK (USPTO)

PEI

| CW_1 | | | DP HEADER | DP PAYLOAD | |

s_i e_i
s_o e_o

FIGURE 8·6

PEI

| CW_1 | | DP HEADER | DP PAYLOAD | | |

s_i e_i
s_o e_o

FIGURE 8·7
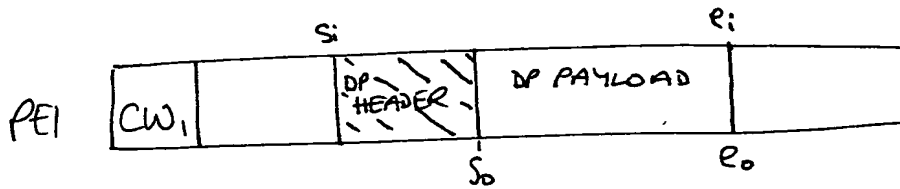
PEI

| CW_1 | | DP HEADER | DP PAYLOAD | |

s_i e_i
s_o e_o
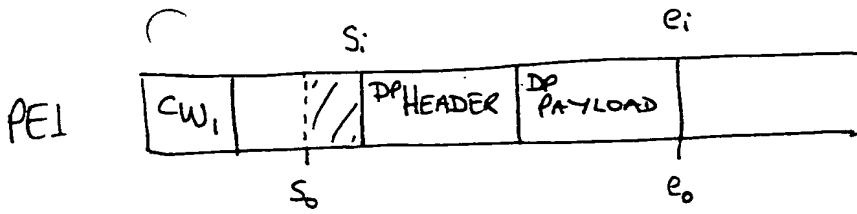
FIGURE 8·8

THIS PAGE BLANK (USPTO)